A Statistically Rigorous Evaluation of the Cascade Bloom Filter for Distributed Access Enforcement in Role-Based Access Control (RBAC) Systems

by

Toufik Zitouni

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2010

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We consider the distributed access enforcement problem for Role-Based Access Control (RBAC) systems. Such enforcement has become important with RBACs increasing adoption, and the proliferation of data that needs to be protected. Our particular interest is in the evaluation of a new data structure that has recently been proposed for enforcement: the Cascade Bloom Filter. The Cascade Bloom Filter is an extension of the Bloom filter, and provides for time- and space-efficient encodings of sets. We compare the Cascade Bloom Filter to the Bloom Filter, and another approach called Authorization Recycling that has been proposed for distributed access enforcement in RBAC. One of the challenges we address is the lack of a benchmark: we propose and justify a benchmark for the assessment. Also, we adopt a statistically rigorous approach for empirical assessment from recent work. We present our results for time- and space-efficiency based on our benchmark. We demonstrate that, of the three data structures that we consider, the Cascade Bloom Filter scales the best with the number of RBAC sessions from the standpoints of time- and space-efficiency.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Mahesh Tripunitara for giving me the chance to be in his research team, and for all the efforts he has done during every phase of the research process, and also for all the support given when we make new discoveries in the research field.

I would like to thank my friend and research partner Marko Komlenovic for providing the access enforcement interface in java that I used to test my own implementation to perform the necessary performance evaluation, and for his positive state-of-mind when working together on a problem.

Portions of this work have been accepted to appear in a peer-reviewed publication [20]

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Access control is the provision of regulated accesses to resources by principals. It is one of the most important aspects of security; indeed, its "center of gravity" [2].

When a user requests access to a resource, an entity called a reference monitor makes a decision (either 'allow' or 'deny') in response. This process is called access enforcement. We present an example of access enforcement in Figure 1.1. In the figure, the user issues requests to read and write a file. The reference monitor mediates the two requests. It allows the user to read the file, and denies the user the ability to write the file. To make its decisions, the reference monitor consults an access control policy. In our example in Figure 1.1, the policy presumably states that the user may read, but not write the file.

We consider access enforcement in the context of Role-Based Access Control(RBAC) [11]. RBAC is a syntax for policies, which is increasingly becoming the de-facto standard in enterprise settings. In RBAC, users are authorized to permissions via roles. In other words, users are assigned to roles, and roles are assigned to permissions. In addition, roles may be assigned to other roles. The relation between roles is called the role-hierarchy. Figure 1.2

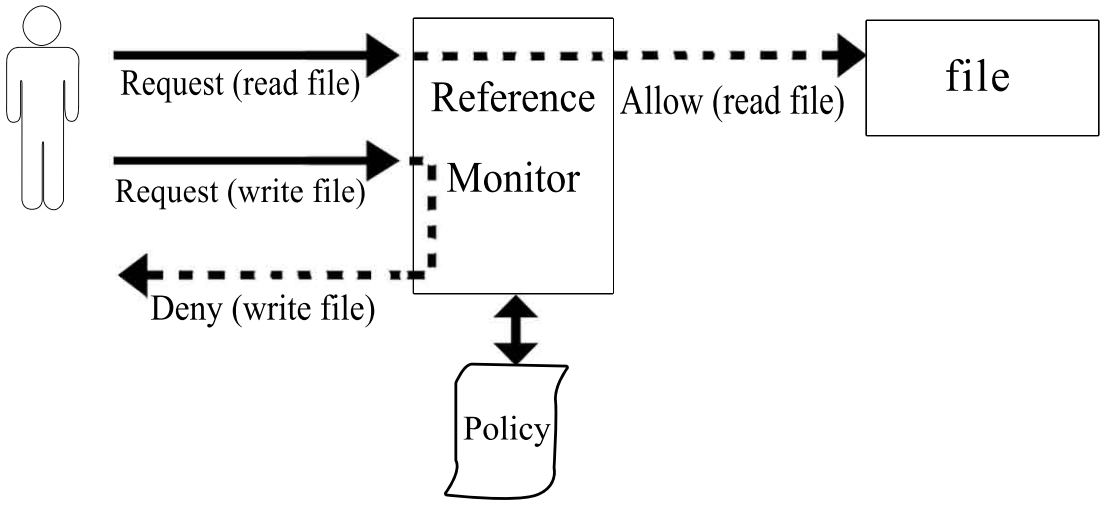


Figure 1.1: An example of access enforcement. Solid lines represent access requests. Dashed lines represent the decision the reference monitor issues in response to access requests.

is an example of an RBAC policy. In the figure, Alice is authorized to the permission *Team Organization* via the role *Project Manager*. As another example, Bob is authorized to the permission *Code Modification* via the role *Software Engineer*, which inherits the role *Developer*.

In RBAC, a user needs to initiate a session to access a resource. A session is associated with a set of roles and provides a context in which a user exercises permissions. A request is allowed if and only if one of the roles associated with the session is authorized to the permission. We say that a role is authorized to a permission if there is a path from the role to the permission in the RBAC policy when it is perceived as a graph. In the RBAC example of Figure 1.2, Alice and Bob may initiate sessions s_a and s_b, respectively. Alice may associate session s_a with the role *Software Engineer*, which authorizes s_a to the permissions *Project Planning* and *Code Modification*. Similarly, Bob may associate session s_b with the roles *Software Engineer* and *IT Consultant*, which authorizes the session s_b to

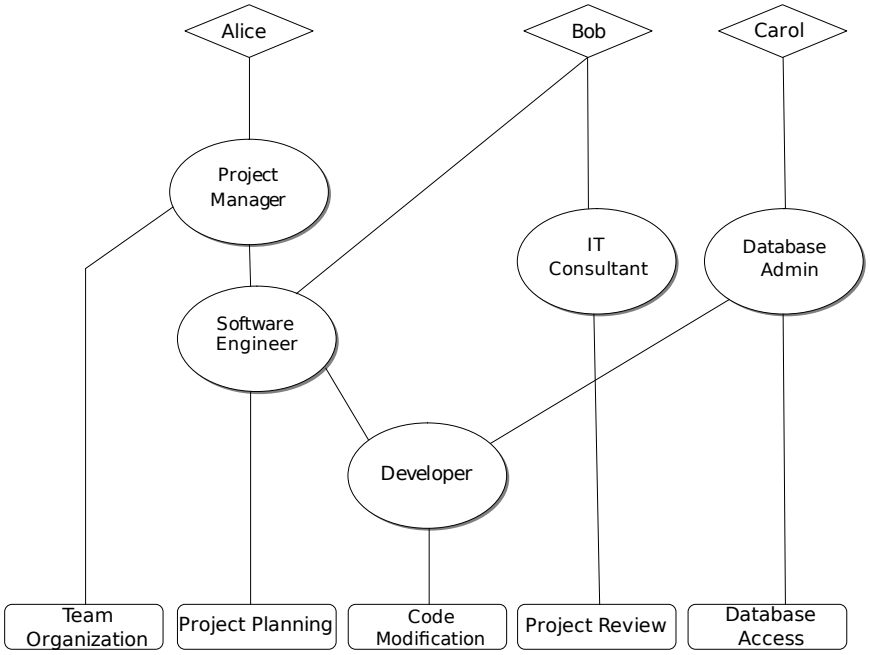


Figure 1.2: An Example RBAC policy. Users are shown in diamonds, roles in ovals and permissions in rectangles. Edges represent user-role, role-role and role-permission assignments. In the example, the user Alice is assigned to the role Project Manager and is therefore authorized to the permission Team Organization. She is also authorized to the role Developer, and therefore to Code Modification.

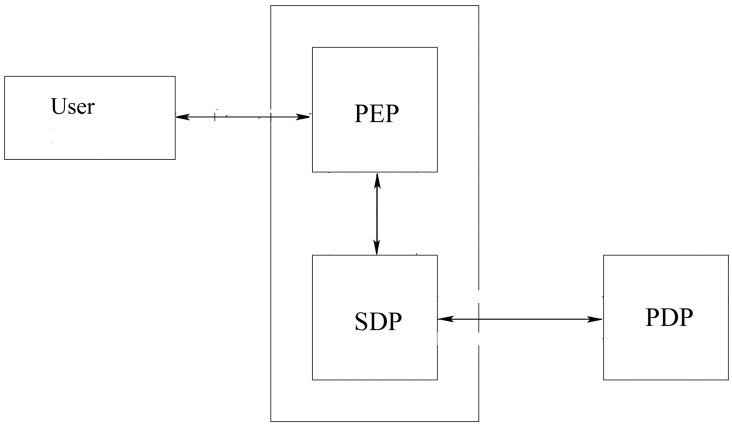


Figure 1.3: Access Enforcement Architecture

the permissions *Project Planning*, *Code Modification*, and *Project Review*.

Modern enterprises generate and archive large amounts of data. The proliferation of data requires access control systems to scale to tens of thousands of resources and permissions [8]. An important aspect of this scalability issue is the time-efficiency of access enforcement. The size of the RBAC policy affects time-efficiency. An approach to the problem is to distribute the reference monitor. With such an approach, a single, monolithic reference monitor is no longer a performance bottleneck. Such distributed enforcement, however, can be at odds with what is touted as one of the main benefits of RBAC — the ease of administration. In RBAC systems, administration comprises changes to one of the three assignments: the user-role, the role-role, and the role-permission assignment. In Figure 1.2, we may assign Alice to the role *IT Consultant*. We may also remove the assignment of Bob to the role *Software Engineer*. Such changes are examples of administration.

To reconcile the issues of time-efficiency and ease of administration, Wei et al. [32] have proposed an architecture for distributed enforcement (see Figure 1.3). In the architecture,

the Policy Decision Point (PDP) is the centralised component which maintains the RBAC policy. The Policy Enforcement Points (PEPs) are distributed components that represent the reference monitor. PEPs are aided by Secondary Decision Points (SDPs). An SDP can be seen as a cache of a portion of the RBAC configuration from the PDP (see Chapter 2).

The question we seek to answer is: what are the data structure and associated algorithms we should use in an SDP? There is evidence that "general purpose" approaches, such as storing an access control policy in a database and using the querying capabilities of the database, do not lend themselves to efficient access enforcement [6]. Consequently, it is necessary to carefully consider the approach that is used.

Two data structures have been proposed in the literature for access enforcement in the context of the architecture in Figure 1.3. They are: Authorization Recycling [32] and the Cascade Bloom Filter [30]. In Authorization Recycling [32], we maintain two sets at the SDP, $Cache^+$ and $Cache^-$. The entries in $Cache^+$ are authorizations that are allowed, and the entries in $Cache^-$ are authorizations that are disallowed. An access request is checked against these sets (see section 2.2).The Cascade Bloom Filter [30] is used for encoding a set A, and checking membership in A (see section 2.1).

Our objective is to provide a sound and reliable performance evaluation of the two data structures in their use for distributed access enforcement. We consider the Bloom Filter [30] in our assessment as well, as the Cascade Bloom Filter is an extension of the Bloom Filter. The Bloom Filter [30] is a probabilistic time- and space-efficient data structure for representing a set (see section 2.1). We address two challenges. One is to provide a meaningful empirical assessment; this is recognized as a challenge in computing [25]. To overcome this challenge, we evaluate the Bloom Filter, the Cascade Bloom Filter, and Authorization Recycling using George et al's approach [13] (see Section 4.1). Another

challenge is the lack of a meaningful benchmark on which to base an assessment. To our knowledge, the only benchmark that has been proposed in the context of RBAC is that of Molloy et al [24]. They provide a benchmark for evaluating role mining algorithms. Their benchmark is insufficient for our purposes as it does not comprise policies that have the size and complexity that we expect in typical enterprise settings. We propose and justify a benchmark that is drawn from prior work on RBAC in three different contexts, and is therefore more comprehensive and realistic (see Chapter 3). Our evaluation shows that the Cascade Bloom Filter is more efficient than the other two data structures for distributed access enforcement in RBAC systems.

1.1 Objectives

The objective of the thesis is to provide a sound and reliable performance evaluation of the Bloom Filter, the Cascade Bloom Filter, and Authorization Recycling. To achieve this objective, we propose and justify a benchmark. We perform a statistically rigorous evaluation of the three data structures. Based on our evaluation, we show that the Cascade Bloom Filter outperforms both Authorization Recycling and the Bloom filter in terms of time- and space-efficiency for distributed access enforcement in RBAC systems.

1.2 Organization of Thesis

The remainder of the thesis is organized as follows. In Chapter 2, we describe the three data structures, their operation, and conduct a literature review. Chapter 3 presents a description of the benchmark used for the evaluation. Chapter 4 presents the results of

the comparison of the three data structures. We conclude with Chapter 5, in which we summarize and discuss future work.

Portions of this work have been accepted to appear in a peer-reviewed publication [20]

Chapter 2

Background

As we discuss in Chapter 1, RBAC is a syntax for policies. An RBAC policy comprises the triple $\langle UA, PA, RH \rangle$ where UA is the user-role assignment, PA is the permission-role assignment, and RH is the role-hierarchy, which is a relation between roles.

There are three RBAC policies that show up in practice. They are: the Stanford Model, the hybrid model, and Core-RBAC. In the Stanford model [11], roles are layered, and a role at layer i directly inherits roles only in layer $i+1$, and is inherited directly only by roles in layer $i-1$ (or by users, for the topmost layer of roles). An example of the Stanford model is shown in Figure 2.1. In the hybrid model, the role hierarchy is some partial ordering, and not layered as in the Stanford model. An example of the hybrid model is shown in Figure 2.2. A special case of the two models is when there is no role-role relationship. This is called Core-RBAC [21, 24]. An example of Core-RBAC is shown in Figure 2.3.

A session is associated with a set of roles and provides a context in which a user exercises permissions. We consider three session operations. They are: session initiation, access request, and session deletion. Session initiation happens when a user associates a

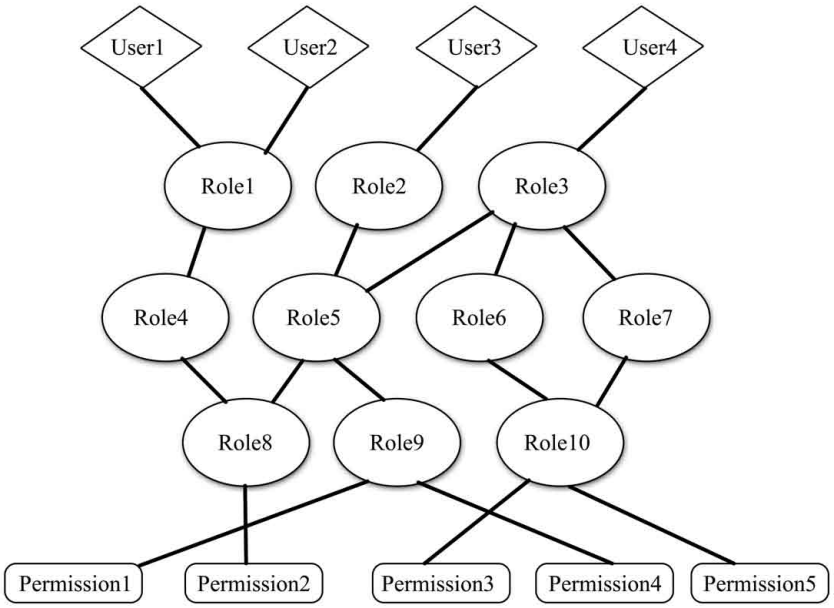


Figure 2.1: Example of a RBAC Stanford model. Users are represented by diamonds, roles by ovals, and permissions by rectangles. A role at layer i directly inherits roles only in layer $i+1$, and is inherited directly only by roles in layer $i-1$. For example, Role1 inherits Role4, but cannot inherit Role8 because Role 8 is not at a layer directly below Role4.

set of roles to a session he initiates. For example, as we discussed in Chapter 1, Alice may associate session s_a with the role *Software Engineer*, which authorizes s_a to the permissions *Project Planning* and *Code Modification*. An access request is generated when a user exercises a permission in the context of a session. For example, in Figure 1.2, Alice may exercise the permission *Project Planning* in the context of s_a. Session deletion is the removal of all information pertaining to the session. For example, removing session s_a results in prohibiting Alice the ability to exercise permissions in the context of s_a.

As we discussed in Chapter 1, the SDP aids the PEP in making access decisions. Also, the SDP can be seen as a cache of a portion of the RBAC configuration from the PDP, which stores the RBAC policy. A user interacts with the architecture in Figure 1.3 by

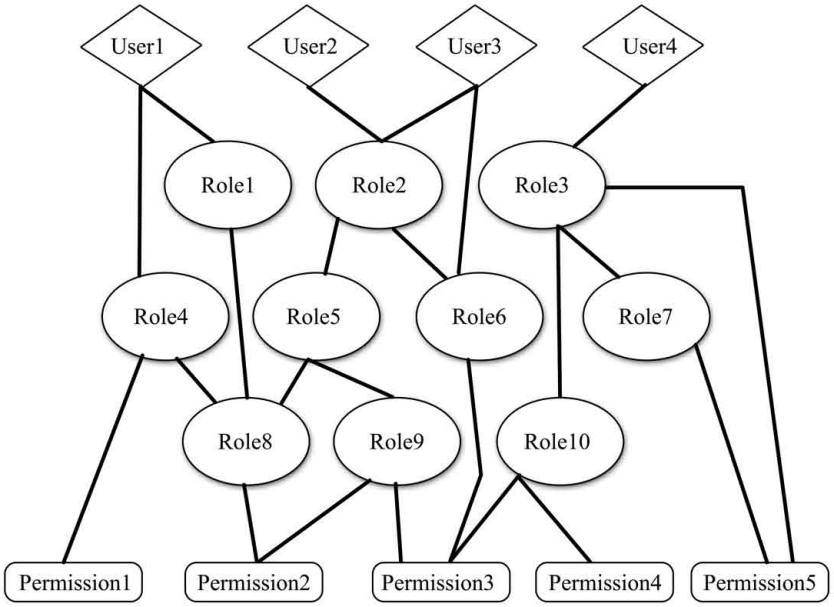


Figure 2.2: Example of a RBAC hybrid model. Users are represented by diamonds, roles by ovals, and permissions by rectangles. A role can inherit any other role in the RBAC policy, or it can inherit a permission. Users can inherit any role. Unlike the Stanford model, there are no layers in the role-hierarchy

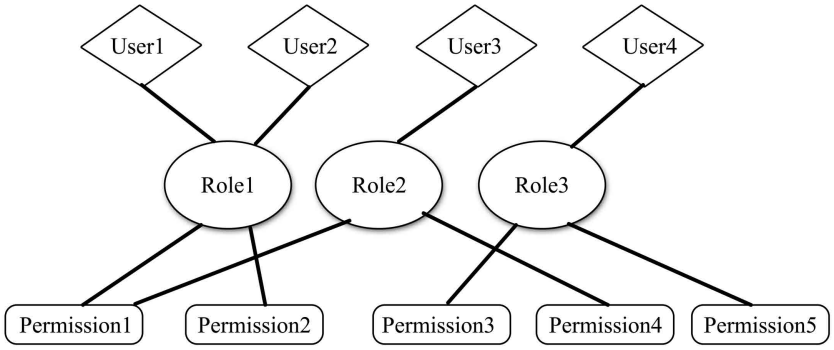


Figure 2.3: Example of a Core-RBAC model. Users are represented by diamonds, roles by ovals, and permissions by rectangles. There is no role-role relationship in this model. Every role inherits a permission, and every user inherits a role.

performing session initiations, access requests, and session deletions. Therefore, the SDP needs to perform the following four operations:

- Construct a data structure to store the data that is communicated from the PDP
- Make access decisions for each access request
- Remove the data associated with a deleted session
- Insert new data that is communicated from the PDP to the already-constructed data structure

The following sections present three data structures that we implement at the SDP. They are: the Bloom Filter (section 2.1), the Cascade Bloom Filter (section 2.1), and Authorization Recycling (section 2.2). Section 2.3 summarizes the work that has been done in research in the context of access enforcement and RBAC systems.

2.1 (Cascade) Bloom Filter

A Bloom filter [4] is a probabilistic time- and space-efficient data structure for encoding a set A, and checking membership of an element e in A. We assume a universe, U, of which A and e are a subset.

A Bloom filter B is an array of m bits, with indices 0 through $m-1$. It is associated with k hash functions $h_1, \ldots, h_k$ each of which maps every $e \in U$ to some integer between 0 and $m-1$.

To represent that $e \in A$, we set the bit to which each h_i maps. To check whether $e \in A$, we check whether the bit to which every h_i maps is set. Figure 2.4 shows an example of

the Bloom Filter operations. In the figure, U is the set of words in a dictionary. $A \subset U$ where $A = \{when, where, what\}$. B is the Bloom Filter with $m = 8$ and $k = 2$. B encodes elements of A. For encoding $when$, we get $h_1(when) = 0$, $h_2(when) = 3$. For $where$: $h_1(where) = 3$, $h_2(where) = 4$. For $what$: $h_1(what) = 5$, $h_2(what) = 7$. We set the bit in B to which every h_i maps. The resulting B is $[1, 0, 0, 1, 1, 1, 0, 1]$. We check for membership of $e = when$ in B. Since all $h_i(e)$ bits are set, e is a member of A. For $e = why$, since bit $h_2(why) = 1$ is not set in B, e is not a member of A.

Removing an element from the Bloom Filter is not as efficient as adding one. Consider the example in Figure 2.4. To remove $e = when$ from B, an intuitive solution would be to reset the bits at every $h_i(when)$. However, resetting bit $h_2(when) = 3$ removes the membership of $e = where$ as well. The only solution is to reconstruct the Bloom Filter B by encoding all elements of A (except $e = when$), which is not efficient. Therefore, a counting Bloom Filter [10] makes removal from a Bloom Filter easier by associating a counter with each index rather than a bit. Figure 2.5 is an example which reimplements the Bloom Filter given in Figure 2.4 by using a counting Bloom Filter. Since $h_2(when)$ and $h_1(where)$ hash to the same slot, the value of B at $h_2(when) = h_1(where)$ is incremented twice. We remove $e = when$ by decrementing every value in B at $h_i(when)$. Since $B[h_2(when)]$ is nonzero after removing e, the membership of $where$ is not compromised.

As k and m are constants in the size of A (and U), B is represented in constant-space, and we can check whether $e \in A$ in constant time. However, this check can result in a false positive; i.e., a check may return 'true' in some cases for which $e \notin A$. For example, in Figure 2.4, a check to $e = who$ where $e \notin A$ may return 'true' if, for example, $h_1(who) = 3$ and $h_2(who) = 5$.Thus, $e = who$ is a false positive. Consequently, a Bloom filter trades-off a probability of false positives for time- and space-efficiency.

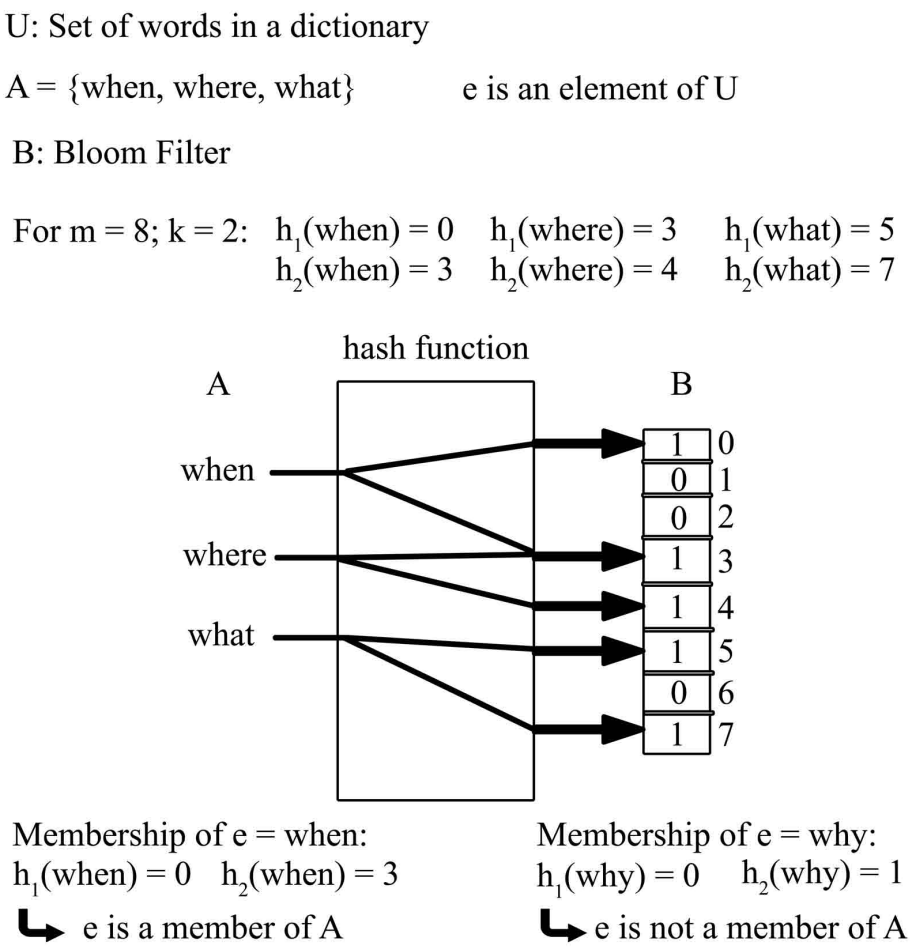


Figure 2.4: Example of the Bloom Filter operation. Set U is the set of words of a dictionary. $A \subset U$ where $A = \{when, where, what\}$. B is the Bloom Filter with $m = 8$ and $k = 2$. B encodes elements of A. $e = when$ is a member of A since the bit in B at $h_1(when) = 0$ and $h_2(when) = 3$ is 1. However, $e = why$ is not a member of A since the bit in B at $h_2(why) = 1$ is 0

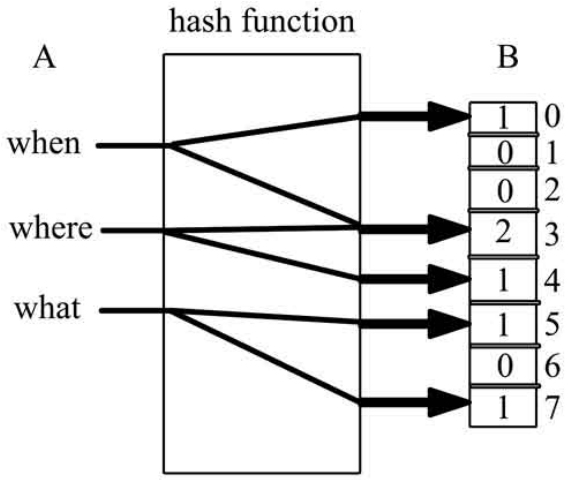


Figure 2.5: Example of the counting Bloom Filter operation, which reimplements the Bloom Filter in Figure 2.4. Since $h_2(when)$ and $h_1(where)$ hash to the same slot, the value of B at $h_2(when) = h_1(where) = 3$ is incremented twice.

The Cascade Bloom Filter [30] is an extension of the Bloom Filter. This data structure uses several Bloom filters and associates each with a level, $l \geq 1$. Let the set encoded by the Bloom Filter at level i, B_i, be called A_i. Then, $A_0 = U$, $A_1 = A$, and for $i > 1$, A_i comprises elements of A_{i-2} that are false positives in B_{i-1}. False positives of B_l are represented as a list.

We illustrate the operations of insertion, membership checking, and deletion performed in the Cascade Bloom Filter with an example. Figure 2.6 shows an example of encoding a set $A = \{when, where, what\}$ in a Cascade Bloom Filter that uses 2 Bloom Filters. The universe U contains the set of words $\{when, where, who, why, whom, what\}$ where $A \subset U$. B_1 is the Bloom Filter at level 1 with $m = 5$ and $k = 2$. B_2 is the Bloom Filter at level 2 with $m = 3$ and $k = 1$. A_1 and A_2 are a subset of U that represent the set of elements that are encoded in B_1 and B_2, respectively.

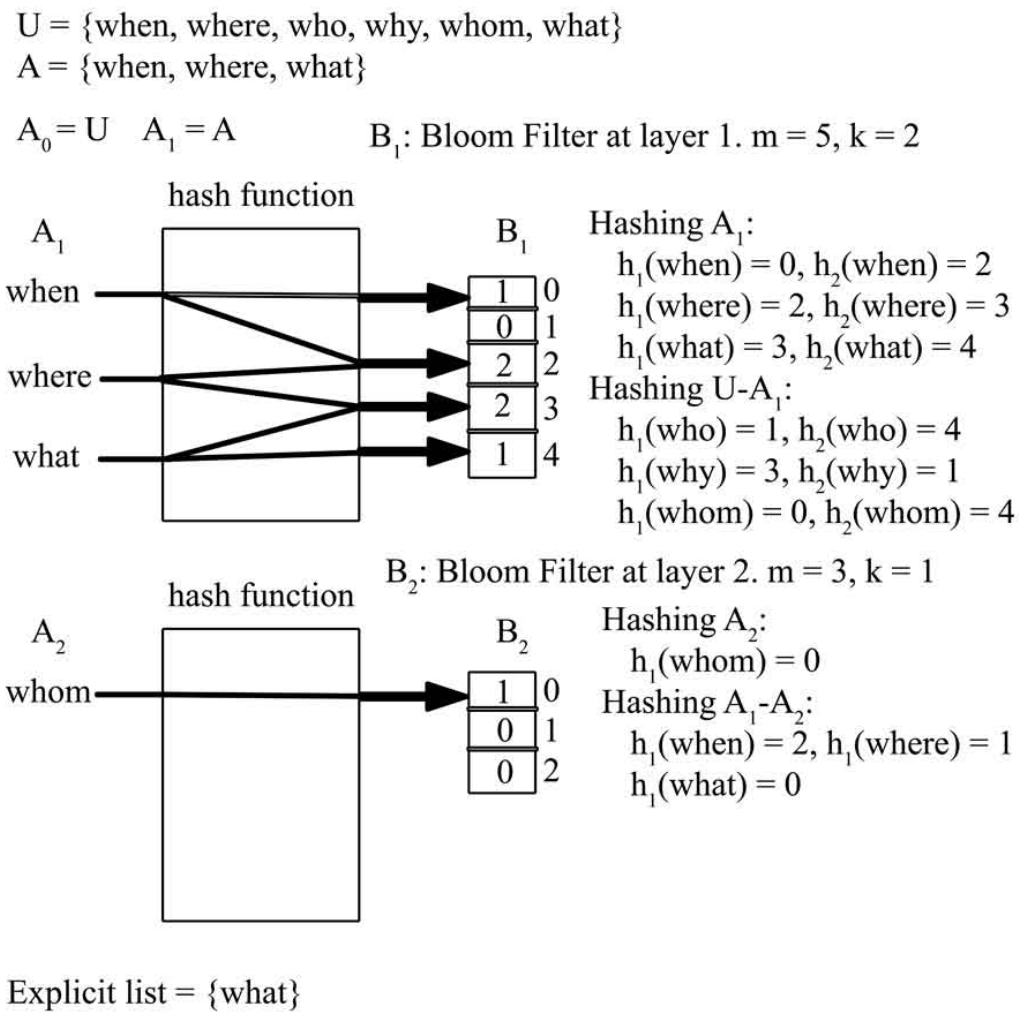


Figure 2.6: Example of a Cascade Bloom Filter with two levels. Set $U = \{when, where, who, why, whom, what\}$. $A \subset U$ where $A = \{when, where, what\}$. B_1 is the Bloom Filter at level 1 with $m = 5$ and $k = 2$. B_2 is the Bloom Filter at level 2 with $m = 3$ and $k = 1$. B_1 encodes elements of A_1 where $A_1 = A$. B_2 encodes elements of A_2 that are false positives of B_1. The explicit list contains the false positives of B_2, which is the last Bloom Filter in the Cascade

Insertion

To encode A in the Cascade, we set $A_1 = A$. The resulting B_1 is $[1, 0, 2, 2, 1]$. We check for false positives in B_1 that may occur from $U - A_1$, where $U - A = \{who, why, whom\}$. Since every h_i of $whom$ hashes to a nonzero value in B_1, $whom$ is a false positive. Therefore, $A_2 = \{whom\}$. We encode A_2 in B_2, resulting in $B_2 = [1, 0, 0]$. We check for false positives that may occur in B_2 from the set $S = A_1 - A_2$. Since every h_i of $what$ hashes to a nonzero value in B_2, $what$ is a false positive. Since the Cascade has 2 levels only, we add the element $e = what$ to the explicit list.

Checking for Membership

We check the membership of two elements: $e = what$, where $e \in A$, and $e = whom$, where $e \notin A$. For $e = what$, we check the membership of e in B_1. Since every h_i of e returns a nonzero value in B_1, e is a member of B_1. We check the membership of e in B_2. Since every h_i of e returns a nonzero value in B_2, e is a member of B_2. Since e is in the explicit list, and the number of levels in the Cascade is an even number, then $e \in A$. For $e = whom$, the membership of e is satisfied in both Bloom Filters. However, e is not in the explicit list. Since the number of levels in the Bloom Filter is even, and e is not in the explicit list, then $e \notin A$.

Deletion

We delete the element $e = what$ where $e \in A$. We remove membership of e from B_1 by decrementing the value of every h_i of e in B_1. The resulting B_1 is $[1, 0, 2, 1, 0]$. We

check for every false positive in B_1 that may no longer be a false positive. $e = whom$ was determined to be a false positive before removing $e = what$. Since $h_2(whom) = 4$ maps to 0 in B_1, $e = whom$ is not a member of the set A. Therefore, $e = whom$ is not a false positive. We remove membership of $e = whom$ from B_2, resulting in $B_2 = [0, 0, 0]$. We remove $e = what$ from the explicit list.

In the next section, we present the algorithms we implement for insertion, checking for membership, and deletion of the Cascade Bloom Filter. We present an algorithm to construct the Cascade Bloom Filter, and we discuss whether a Cascade Bloom Filter exists given the set A and U and the parameters m and k.

2.1.1 Algorithms

Algorithm 1 presents the algorithm taken from [30] for inserting an element into a Cascade Bloom Filter. We note the following method:

InsertIntoCascadeBF(level, I, U') – The algorithm takes as input the set I, representing the set to be added to the Cascade Bloom Filter. The set U' is the new universe after adding I, where $U' = U + I$. *level* is the level of the Bloom Filter in the Cascade. For each invocation, the method processes two levels at a time, an odd level and the next even one. For each level, we maintain a set, A_i, that contains the elements that are encoded in the Bloom Filter at level i. I is added to A_{level} and encoded in BF_{level} in lines 24-29. After adding I in lines 24-29, we consider the next even level. In this level, we add false positives resulting from elements added in the odd level, and we remove false positives added previously that are no longer false positives. The operation is done in lines 32-63. The algorithm recurses in lines 64-76

to add elements to the remainder levels of the Cascade with the corresponding I and U'. The explicit list adds the final false positives in lines 15-22.

Algorithm 2 presents the algorithm taken from [30] for checking for membership of an element in a Cascade Bloom Filter. We note the following method:

MemberCascadeBF(e) – This method takes as input the element e to be checked for membership in the Cascade Bloom Filter. In Lines 7-15, we iterate through each level of the Cascade. If, any any level, e returns $false$ for its membership in the Bloom Filter, we can make an inference about the membership of e in A. If the level is even, then $e \in A$. Otherwise, $e \notin A$. If e returns $true$ for its membership in each Bloom Filter of the Cascade, we check the explicit list E holding the final false positives. From lines 16-28, if the depth of the Cascade is even, and $e \in E$, then $e \in A$. If $e \notin E$, then $e \notin A$. Otherwise, if the depth of the Cascade is odd, and $e \in E$, then $e \notin A$. If $e \notin E$, then $e \in A$

Algorithm 3 presents the algorithm for removing an element from a Cascade Bloom Filter. We note the following method:

RemoveFromCascadeBF(U') – This method removes elements in the set U' from the Cascade. We remove all elements in U' from the explicit list in lines 5-9. In lines 10-17, we remove membership of each element e in U' from all Bloom Filters in the Cascade, and we remove membership of elements that are no longer false positives from the Cascade in lines 18-30.

Algorithm 4 presents the algorithm taken from [30] for creating a Cascade Bloom Filter. We note the following methods:

Algorithm 1 An algorithm for inserting elements into a Cascade Bloom Filter

Operation InsertIntoCascadeBF(level, I, U')

```
1: // Insert the elements of I into a Cascade Bloom Filter
2: // that represents A from universe U.
3: // The set U' ⊇ U is the new universe
4: // It is first invoked with level = 1
5: // d is the depth of the Cascade Bloom Filter
6: // E is the explicit list
7:
8: if level is even or level > d + 1 then
9:     return false
10: end if
11: if I ⊈ U' then
12:     return false
13: end if
14: // If this layer is the last layer
15: if level = d + 1 then
16:     for all e ∈ I do
17:         if e ∉ E then
18:             add e to E
19:         end if
20:     end for
21:     return true
22: end if
23: // Add I to this layer
24: for all e ∈ I do
25:     if e ∉ A_level then
26:         add e to A_level
27:         encode e in BF_level
28:     end if
29: end for
30: // Now do the next level as well
31: // If this is the last layer
```

```
32: if level = d then
33:     for all e ∈ I do
34:         if e ∈ E then
35:             remove e from E
36:         end if
37:     end for
38:     // Consider new false positives
39:     F = U' - (A_level ∪ E)
40:     for all e ∈ F do
41:         if e is encoded in BF_level then
42:             add e to E
43:         end if
44:     end for
45:     return true
46: end if
47: // Remove false positives that are no longer so
48: for all e ∈ I do
49:     if e ∈ A_level+1 then
50:         remove e from A_level+1
51:         remove e from BF_level+1
52:     end if
53: end for
54: // Add new false positives
55: F = U' - (A_level ∪ A_level+1)
56: for all e ∈ F do
57:     if e tests positive for membership in BF_level+1 then
58:         if e ∉ A_level+1 then
59:             add e to A_level+1
60:             encode e in BF_level+1
61:         end if
62:     end if
63: end for
64: // Prepare for recursion
65: I' = 0
66: Generate set to be inserted in A_level+2
67: for all e ∈ A_level do
68:     if e tests positive for membership in BF_level+1 then
69:         I' = I'∪e
70:     end if
71: end for
```

```
U" = I'∪ A_{level+1}
// Recurse
if I' ≠ 0 or U" ≠ 0 then
    InsertIntoCascadeBF(level + 2, I', U")
end if
```

Algorithm 2 An algorithm to verify membership of an element e in a set encoded in the Cascade Bloom Filter

Operation MemberCascadeBF(e)

```
// BF is the Bloom Filter
// level is the level at which a BF is located in the Cascade
// E is the explicit list storing the final false positives of the
// cascade
// d is the depth of the Cascade Bloom Filter

for level = 1 to d do
    if e tests positive for membership in BF_{level} then
        if level is even then
            return true
        else
            return false
        end if
    end if
end for
if d is even then
    if e ∈ E then
        return true
    else
        return false
    end if
else
    if e ∈ E then
        return false
    else
        return true
    end if
end if
```

Algorithm 3 An algorithm to remove membership of a set of elements from the Cascade

Operation RemoveFromCascadeBF(U')

```
1: // U' represents the set of elements to be removed from the Cascade
2: // E is the explicit list holding the final false positives
3: // d is the depth of the Cascade
4:
5: for all e ∈ U' do
6:     if e ∈ E then
7:         remove e from E
8:     end if
9: end for
10: for level = 1 to d do
11:     for all e ∈ U' do
12:         if e ∈ A_level then
13:             remove e from A_level
14:             remove membership of e from BF_level
15:         end if
16:     end for
17: end for
18: for level = 1 to d do
19:     for all e ∈ A_level+1 do
20:         if e tests negative for membership in BF_level then
21:             remove e from A_level+1
22:             remove membership of e from BF_level+1
23:         end if
24:     end for
25: end for
```

ConstructCascadeBF(A, U, m, Elen) – This method takes as input the set A to be encoded, the universe U, the maximum number of counters for the Bloom Filters m, and the maximum size of the explicit list $Elen$, which stores the final list of false positives. The method invokes **tryToConstruct**, which attempts to create a Cascade Bloom Filter with the given inputs mentioned previously.

tryToConstruct(level, depth, Asize, Usize, m, Elen) – This method tries to construct a Bloom Filter at a specific level. It calculates the size of the Bloom Filter in line 8, and the number of hashes in line 9. If $newAsize \leq Elen$ in line 13, then the algorithm has successfully identified a Cascade Bloom Filter that can be constructed, which is done by executing lines 14-16.

Existence of a Cascade Bloom Filter

Let $L = \{\langle U, A, m, e\rangle$: there exists a Cascade Bloom Filter that encodes $A \subseteq U$ with m bits and an explicit list of size at most $e\}$. We point out that it is likely that $L \notin \mathbf{NP}$. The reason is that the most natural certificate for an instance of L is a corresponding (instance of a) Cascade Bloom Filter, which may have up to $\Theta(m)$ levels. As the input m can be represented in $\log m$ bits, the resultant certificate is not polynomial in the size of the input.

We may assume, however, that the m bits and a list of size e are pre-allocated. We correspondingly change our characterization of L to $\{\langle U, A, m, e\rangle$: given m bits, a list E with $|E| = e$, and $A \subseteq U$, there exists a Cascade Bloom Filter that encodes A with those m bits and $E\}$. Now, given some input, an algorithm has only to partition the m bits into levels. We assume further that at each level of the Cascade Bloom Filter, the number of hash functions is at most the number that minimizes the false positive rate for the Bloom

Algorithm 4 An algorithm for creating a Cascade Bloom Filter

Operation ConstructCascadeBF(A, U, m, Elen)

```
1: // m is the total number of bits for Bloom Filters
2: // Elen is the maximum length of the explicit set E
3: // MAX_DEPTH is the total depth of the Cascade (=1 for Bloom Filter)
4:
5: for depth = MAX_DEPTH to 1 do
6:     if tryToConstruct(1, depth, |A|, |U|, m, Elen) then
7:         A_0 = U
8:         A_1 = A
9:         for level = 1 to depth do
10:            for all e ∈ A_level do
11:                encode e in BF_level
12:            end for
13:            for all e ∈ A_{level-1} − A_level do
14:                if e is a member of BF_level then
15:                    if level = depth then
16:                        add e to E
17:                    else
18:                        encode e in BF_{level+1}
19:                        add e to A_{level+1}
20:                    end if
21:                end if
22:            end for
23:        end for
24:    end if
25: end for
26: return false
```

Operation tryToConstruct(level, depth, Asize, Usize, m, Elen)

```
1: // Try to construct a Bloom Filter at level and below
2:
3: if m ≤ 0 then
4:     return false
5: end if
6: trials = ⌊m/Asize⌋ − 1
```

```
7: for i = trials − 1 to 0 do
8:     m_i = (⌊m/Asize⌋ − i) × Asize
9:     k_i = round(ln2 × m_i/Asize)
10:    ε_i = (1 − e^{−k_i × Asize/m_i})^{k_i}
11:    newAsize = ⌈ε_i × (Usize − Asize)⌉
12:    if level = depth then
13:        if newAsize < Elen then
14:            construct Bloom Filter of m_i bits and k_i hashes
15:            at level; set | E | = newAsize
16:            return true
17:        end if
18:    else
19:        if tryToConstruct(level + 1, depth, newAsize, newAsize + Asize, m - m_i, Elen)
       then
20:            construct Bloom Filter of m_i bits and k_i hashes
21:            at level
22:            return true
23:        end if
24:    end if
25: end for
26: return false
```

filter at that level. This is reasonable, as there are no benefits to using more hash functions at a level.

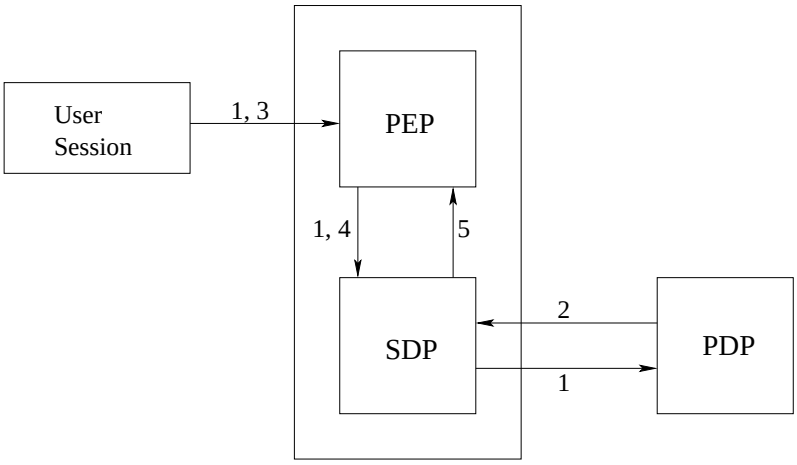
For level i of the Cascade, let the number of hash functions be k_i. Then we know that for a level-l, $l \leq m$, and $\sum_{i=1}^{l} k_i \leq \sum_{i=1}^{l} \frac{m_i}{|A_i|} \ln 2 < m$, where m_i is the number of bits allocated to level i, and A_i is the set encoded by the Bloom Filter at level i. Now, given some $l = \langle U, A, m, e \rangle \in L$, and a certificate c for it that is a Cascade Bloom Filter, we know that c is polynomial (linear) in the size of the input. Furthermore, an algorithm that verifies that c is indeed a certificate for l simply checks that for every $a \in U$, the Cascade returns 'true' if and only if $a \in A$. For each $a \in U$, this algorithm runs in time $O(|A| + m + e)$, and therefore the entire algorithm is quadratic in the size of the input. We conclude that $L \in \mathbf{NP}$.

The exact complexity-class in which L lies is unknown as of the writing of this thesis.

2.1.2 Application in Access Control

Figure 2.7 shows a chronological process-flow for distributed access-enforcement in RBAC. In Step 1, a user initiates a session at a PEP/SDP. The request to activate a session propagates to the PDP, which makes the decision on whether it is allowed. If it is, in Step 2, the PDP communicates a data structure to the SDP that the latter uses in Steps 3, 4 and 5 to make decisions on access requests that pertain to that session, that are communicated to it by the PEP.

The encoding of RBAC sessions in a Cascade Bloom Filter that has been proposed [30] is as follows. Let $S = \{s_1, \ldots, s_n\}$ be the set of active sessions at a PEP-SDP, and P be the set of permissions such that $p \in P$ if any $s_i \in S$ is authorized to P. Then, U



1: Session initiation request
2: Access enforcement structure
3: Access request
4: Validated/translated access request
5: Access decision

Figure 2.7: An architecture, reproduced from prior work [30, 32], for distributed access-enforcement in RBAC, and an associated process-flow. Our focus is on the Access enforcement (data) structure that is received by the SDP in Step 2, and that it uses to make access decisions.

is $S \times P$, and A is the smaller (in cardinality) of $A_p = \{\langle s, p\rangle : s \text{ is authorized to } p\}$ and $A_n = \{\langle s, p\rangle : s \text{ is not authorized to } p\}$. A negation bit indicates which of A_p or A_n the Cascade Bloom Filter encodes. The Bloom Filter is a special case of the Cascade Bloom Filter, with the number of levels, $l = 1$. We adopt the same encoding of RBAC sessions for the Bloom Filter as the Cascade Bloom Filter.

2.1.3 Example

Figure 2.8 is an example of using the Cascade Bloom Filter. In the figure, Alice and Bob initiate sessions $\langle s_a, \{$*Software Engineer*$\}\rangle$ and $\langle s_b, \{$*Software Engineer, IT Consultant*$\}\rangle$, respectively. Therefore, $U = \{s_a, s_b\} \times \{$ *Project Planning, Code Modification, Project Review*$\}$, $A_p = \{\langle s_a,$ *Project Planning* $\rangle$, $\langle s_a,$ *Code Modification* $\rangle$, $\langle s_b,$ *Project Planning* $\rangle$, $\langle s_b,$ *Code Modification* $\rangle$, $\langle s_b,$ *Project Review* $\rangle\}$, $A_n = U - A_p$. Since $|A_p| > |A_n|$, the Cascade Bloom Filter contains the set of negative authorizations (the negation bit is set).

The Cascade Bloom Filter has 2 levels. Bloom Filter B_1 is at level 1 with $m = 5$ and $k = 3$. Bloom Filter B_2 is at level 2 with $m = 3$ and $k = 2$. We encode A_n in B_1, which results in $B_1 = \{1, 0, 2, 0, 0\}$. We check for false positives in B_1. Since B_1 is nonzero at every $h_i(e)$ where $e = \{\langle s_b,$ *Project Review* $\rangle\}$, then e is a false positive of B_1. We encode e in B_2, which results in $B_2 = \{0, 2, 0\}$. We check for false positives in B_2. Since B_2 is nonzero at every $h_i(e)$ where $e = \{\langle s_a,$ *Project Review* $\rangle\}$, then e is a false positive of B_2. Therefore, the explicit list $A_3 = \{\langle s_a,$ *Project Review* $\rangle\}$.

If Alice exercises the permission *Project Review* in the context of s_a, we check the membership of $e = \{\langle s_a,$ *Project Review* $\rangle\}$ in every Bloom Filter of the Cascade. Since all Bloom Filters return a nonzero value, we check the explicit list. Since the Cascade Bloom

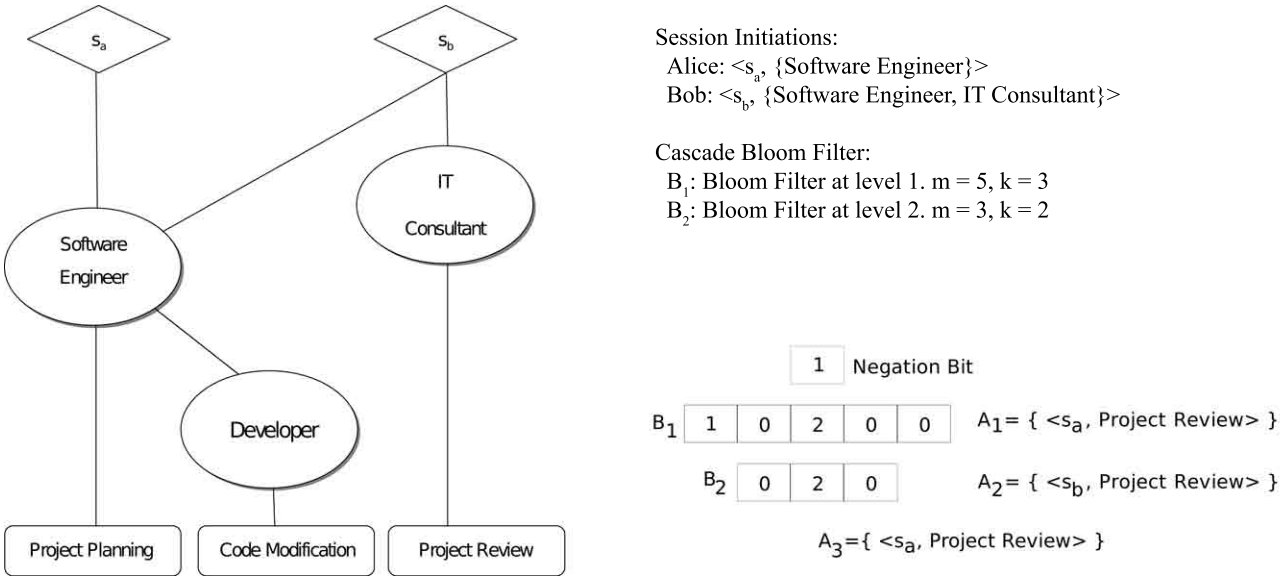


Figure 2.8: Example of using a Cascade Bloom Filter with 2 levels for sessions s_a and s_b initiated by Alice and Bob, respectively, for the RBAC policy in Figure 1.2. We assume that three hash functions are used for Level 1, and two are used for Level 2. The Negation Bit is set, which indicates that A_1 is the set of authorizations that are disallowed. A_2 is the set of false positives in B_1, and $A_3 \subseteq A_1$ is the set of false positives in B_2, which is the explicit list.

Filter contains an even number of layers and the negation bit is set, $e \notin A$. Therefore, the access is disallowed.

2.2 Authorization Recycling

In Authorization recycling [32], we maintain two sets, Cache$^+$ and Cache$^-$. Cache$^+$ stores the tuple $\langle R, p \rangle$ where R represents the set of roles that are authorized to the permission p. Cache$^-$ stores one entry $\langle R, p \rangle$ where every $r \in R$ is not authorized to the permission p.

2.2.1 Application in Access Control

Figure 2.9 shows a chronological process-flow for distributed access-enforcement in RBAC. In Step 1, a user initiates a session at a PEP/SDP. The user requests a permission in step 2. The request may propagate to the PDP if the SDP cannot make an access decision. The PDP communicates a data structure to the SDP in step 3 that the latter uses to make decisions on access requests in steps 2, 4 and 5. The SDP updates the two sets it maintains when it receives the data structure from the PDP.

In Figure 2.9, access enforcement is performed as follows. When a user exercises a permission p, the SDP performs one of the following actions:

- Returns *Denied*.
- Returns *Allowed*.
- The SDP is *Undecided*. It requests more information from the PDP to make an access decision and updates $Cache^+$ and $Cache^-$.

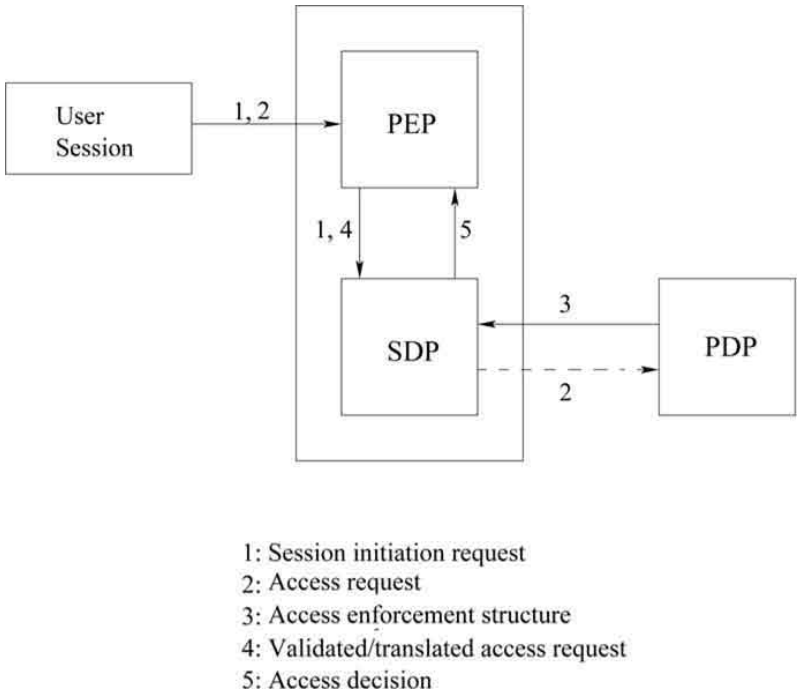


Figure 2.9: An architecture, reproduced from prior work [30, 32], for distributed access-enforcement in RBAC, and an associated process-flow. A request propagates to the PDP only when the SDP cannot make an access decision in step 2. This is shown as a dashed line. Our focus is on the Access enforcement (data) structure that is received by the SDP in Step 3, and that it uses to make access decisions. The difference between this and Figure 2.6 is that the SDP communicates with the PDP when an access request is made, instead of at session initiation.

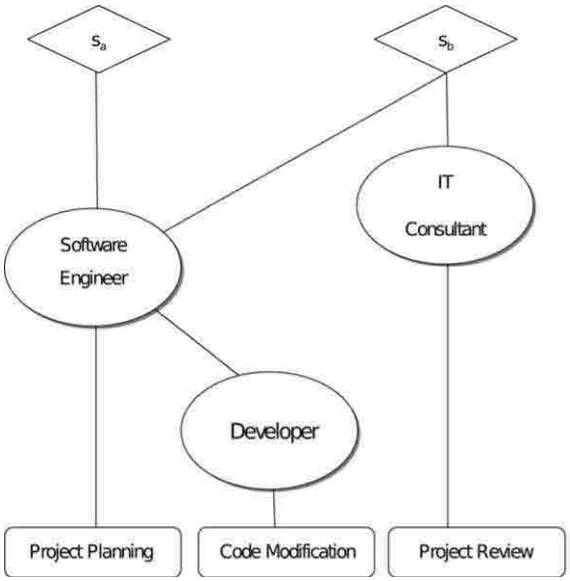
When a user exercises the permission p in the context of a session where a set of roles R has been specified, the SDP returns *Denied* if $R \subseteq R^-$ where $\langle R^-, p\rangle \in Cache^-$. Likewise, the SDP returns *Allowed* if $R \supseteq R^+$ where $\langle R^+, p\rangle \in Cache^+$. If neither of the previous conditions is satisfied, the SDP cannot make a final decision. Therefore the SDP requests more information from the PDP to make an access decision and update $Cache^+$ and $Cache^-$.

2.2.2 Algorithms

Algorithm 5 is an algorithm taken from [32] for performing access decisions by the SDP when a user exercises a permission. We note the following methods:

EvaluateRequest(r, p) – This method takes as input the set of roles r the user specifies at session initiation and the permission p he exercises. The access request is denied if $r \subseteq R^-$ where $\langle R^-, p\rangle \in Cache^-$ (lines 6-7). The request is allowed if $r \supseteq R^+$ where $\langle R^+, p\rangle \in Cache^+$ (lines 8-9). If no decision can be made, the SDP first communicates with the PDP, and then makes a decision. The method **AddResponse(r, p)** updates the sets maintained by the SDP (lines 10-11).

AddResponse(r, p) – This method is called by the SDP to update its sets when it is undecided about the access request. It takes as input the set of roles r the user specifies at session initiation, and the permission p he exercises in the context of a session. In lines 1-7, if none of the roles the user specifies at session initiation is authorized to the permission he exercises, the roles are added to Cache$^-$ and each of those roles is removed from the entry in Cache$^+$ where it is authorized to p. In lines



Roles:
SE: Software Engineer
ITC: IT Consultant

Permissions:
TO: Team Organization
CM: Code Modification
PR: Project Review

Session Initiations:
$<s_a, \{SE\}>$
$<s_b, \{SE, ITC\}>$

Requests:
1. s_a: CM
2. s_a: TO
3. s_b: CM

Request	*Cache*$^+$	*Cache*$^-$
({SE}, CM)	({SE}, CM)	
({SE}, TO)	({SE}, CM)	({SE}, TO)
({SE, ITC}, CM)	({SE}, CM)	({SE}, TO)

Figure 2.10: Example of access enforcement using Authorization Recycling. Alice and Bob initiate sessions $\langle s_a$, $\{$*Software Engineer*$\}\rangle$ and $\langle s_b$, $\{$*Software Engineer*, *IT Consultant*$\}\rangle$, respectively. The order of access requests is as follows: ($\{$*Software Engineer*$\}$, *Code Modification*), ($\{$*Software Engineer*$\}$, *Team Organization*), and ($\{$*Software Engineer*, *IT Consultant*$\}$, *Code Modification*). *Cache*$^+$ and *Cache*$^-$ are updated for each request. Alice and Bob are allowed to exercise the permission *Code Modification*, but Alice is not allowed to exercise the permission *Team Organization*

8-12, if any role in r is authorized to p, each role is added to Cache$^+$ if and only if there is no entry in Cache$^-$ that contains a mapping of that role to p.

RemoveSession(s) – This method is called when a session s is deleted. The method takes session s as input. R is the set of roles associated with s. In lines 3-11, if $r \in R$ and r was activated by session s only, then r is removed from Cache$^+$ or Cache$^-$.

Algorithm 5 An algorithm for making access decisions and updating Cache$^+$ and Cache$^-$ when an access request or a session deletion is performed using Authorization Recycling

Operation EvaluateRequest(r, p)

```
1: // r is the set of roles a user specifies when he initiates a session
2: // p is the permission the user exercises
3: // R+ is the set of roles in Cache+ that are authorized to p
4: // R− is the set of roles in Cache− that are not authorized to p
5:
6: if r ⊆ R−, where (R−, p) ∈ Cache− then
7:     return deny
8: else if there exists (R+, p) ∈ Cache+ s.t. R+ ⊆ r then
9:     return allow
10: else
11:     the SDP is undecided. The SDP communicates with the PDP to finalise its
12:     decision and updates its sets
13: end if
```

Operation AddResponse(r, p)

```
1: if all roles in r are not authorized to p then
2:     replace each (R+, p) ∈ Cache+ with (R+ - r, p)
3:     if (R−, p) ∈ Cache− then
4:         replace it with (r ∪ R−, p)
5:     else
6:         add (R, p) to Cache−
7:     end if
8: else
9:     find (R−, p) ∈ Cache−
10:    delete all (R+, p) ∈ Cache+ s.t. r - R− ⊆ R+
11:    add (r - R−, p) to Cache+
12: end if
```

Operation RemoveSession(s)

```
1: // s is the session initiated
2: // R is the set of roles associated with session s
3: for all r ∈ R do
4:     if r was activated by session s only then
5:         if r ∈ Cache+ then
6:             remove r from Cache+
7:         else
8:             remove r from Cache−
9:         end if
10:    end if
11: end for
```

2.2.3 Example

Figure 2.10 shows an example of using Authorization Recycling for access enforcement. In the figure, Alice and Bob initiate sessions $\langle s_a$, $\{$*Software Engineer*$\}\rangle$ and $\langle s_b$, $\{$*Software Engineer*, *IT Consultant*$\}\rangle$, respectively. Alice requests the permissions *Code Modification* and *Team Organization* in the context of s_a, and Bob requests the permission *Code Modification* in the context of s_b. Since no information is available in $Cache^+$ and $Cache^-$ for the first request, the information is retrieved from the PDP, $\langle\{$*Software Engineer*$\}$, *Code Modification* $\rangle$ is added to Cache$^+$, and the request is allowed. For the second request, no information is available in both sets for the permission *Team Organization*. After consulting with the PDP, the SDP adds $\langle\{$*Software Engineer*$\}$, *Team Organization* $\rangle$ to Cache$^-$. For the final request, Bob is allowed to exercise the permission *Code Modification*, since $\{$*Software Engineer*, *IT Consultant*$\} \subseteq \{$*Software Engineer*$\}$.

2.3 Related Work

There is a large amount of research in distributed access control, and in distributed RBAC in particular. However, there is relatively little work on efficient access enforcement in these contexts. To our knowledge, CPOL [6] is the state of the art in access enforcement in distributed settings. CPOL employs caching and a structure called an AccessToken that is application-specific to speed-up access enforcement. The work on CPOL points out also that simply using database querying does not suffice for fast access enforcement. Our work is close also to those of Wei et al. [32], Tripunitara and Carbunar [30] and Liu et al. [21], that address the access enforcement problem in RBAC. Wei et al. [32] propose the architecture that we adopt in this paper (see Figure 1.3). In that context, they propose

Authorization Recycling which is one of the data structures that we assess. Tripunitara and Carbunar [30] adopt the architecture of Wei et al. [32] and propose an approach called the Cascade Bloom Filter for access checking. Their focus is fast and space efficient access checking for RBAC in low-capability devices. Liu et al. [21] propose a technique that they call transformations for access checking in RBAC.

Chapter 3

Categorization of RBAC Policies and Session Profiles in our Benchmark

We have devised a benchmark for access enforcement in RBAC systems. We categorize two components in our benchmark: RBAC policies (section 3.1) and session profiles (section 3.2). We present datasets for RBAC policies and session profiles.

3.1 RBAC Policy

We generate and create new RBAC policies based on prior work and experience with RBAC deployments that have been documented in books and the research literature. We present a summary of datasets for RBAC policies in Table 3.1. We categorize RBAC policies along the following axes:

Source We have two sources of datasets: "Literature", and "Synthetic". Datasets from Literature are documented works of RBAC policies that researchers have used in their

Source	# Users	# Roles	# Permissions	RH Depth	RH Model	Connectivity
Literature	500-999	10-200	10-3000	1–5	Stanford Hybrid Core	Constant (range) to to roles, Constant (range) to permissions, Distributions (e.g., Zipf, uniform)
	1000-1999	200	1000-3500			
	2000-2999	100	100-2000			
	3000-3999	200-250	1500-11000			
	5000-6000	200	1500-2000			
	10000-40000	120-1300	100-11000			
Synthetic	40001-400000	1600-16000	1500-2000			
	400001-1600000	16000-64000	1500-11000			

Table 3.1: RBAC Policy Categorization.

publications. We classify our sources from Literature into three categories:

1. Top-down design of RBAC policies [11, 17, 18, 27]
2. Role mining and engineering [5, 12, 15, 23, 24, 31, 34, 35]
3. Evaluation of approaches to access-enforcement [21, 30, 32]

We create new RBAC policies based on ones from the literature, which we call Synthetic policies.

Number of users, roles and permissions The numbers of users, roles and permissions are typically co-dependant in RBAC policies from the literature. We point out the following observations from Table 3.1:

- The number of roles grow as a step function with respect to the number of users.
- The number of permissions range from a fraction of the number of users to a somewhat significant multiple.

The second observation stems from the fact that RBAC is deployed in one of two contexts. One is for high-level policies in which permissions are abstract. Another is at a much lower

level, in which resources that are protected are individual files or email messages; in such systems, there can be a considerable number of permissions. (It is common for a permission to be a pair $\langle o, r \rangle$, where o is the object or resource that is protected, and r is a privilege or right. However, this is not the only encoding as a permission that is meaningful; see, for example, the work of Crampton [9].)

We consider Synthetic datasets that have not been considered in Literature. Literature datasets extend to 40000 users only. However, an enterprise can have up to 1.6 million employees [1]. We anticipate that such enterprises will want to model each employee as an RBAC user. We also anticipate that the number of roles will be in the same proportion to the number of users as for the largest range for users from the literature. Therefore, our Synthetic dataset includes users up to 1.6 million where they can be assigned to the same proportion of roles as described in the Literature source. We do not anticipate, however, that the number of permissions will increase significantly. Consequently, we adopt for permissions similar numbers as the largest ranges from the literature.

Role Hierarchy (RH) and connectivity In Table 3.1, we consider three categories for the structure of RBAC policies. They are: RH Depth, RH Model, and Connectivity. RH Depth is the number of layers in the role-hierarchy. In our survey of the literature, the RH Depth does not exceed 5.

We consider three RH Models: Stanford, Hybrid and Core. In the Stanford model [11], roles are layered, and a role at layer i directly inherits roles only in layer $i + 1$, and is inherited directly only by roles in layer $i - 1$ (or by users, for the topmost layer of roles). The Stanford model arises in the top-down design of RBAC policies. Realizing the Stanford model in an enterprise typically results in 4 or 5 layers of roles [11]. The hybrid model arises in both the top-down design of RBAC policies and in role mining. In the hybrid model,

the role hierarchy is some partial ordering, and not layered as in the Stanford model. A special case of the two models is when there is no role-role relationship. This is called Core RBAC and arises in role mining [21, 24].

Algorithm 6 is an algorithm we have designed and implemented to generate RBAC policies for our benchmark. We note the following method:

RBACPolicy(nu, nr, np, dr, nrh, rc, uc, pc) – this method takes as input the number of users (nu), roles (nr), and permissions (np), the depth (dr) and nature (nrh) of the role-hierarchy, the role (rc), user (uc), and permission (pc) connectivity. In line 15, we divide the roles into layers according to dr and each layer has nr/dr roles. Lines 18-60 construct the RBAC policy by connecting each element according to the criteria specified at the inputs. Users, roles, and permissions can be connected either randomly, or uniformly, in a Stanford style model, or Hybrid. If the depth is 1, then the RBAC model is Core.

3.2 Session Profile

There is some prior work which has datasets on session profiles [21, 30, 33]. We augment those datasets with our own.

We consider three categories in our session profiles: sessions, role activation, and access requests; we summarize these categories in Table 3.2.

Sessions The total number of session initiations can vary from a fraction of the number of users to a multiple since a user may initiate more than one session. However, the number of active sessions is less than or equal to the total number of session initiations. By active

Algorithm 6 An algorithm for generating different RBAC policies

Operation RBACPolicy(nu, nr, np, dr, nrh, rc, uc, pc)

```
1: // nu is the total number of users represented in the RBAC policy
2: // nr is the total number of roles in the RBAC policy
3: // np is the total number of permissions in the RBAC policy
4: // dr is the depth of the role-hierarchy. It is 1 for Core-RBAC and
5: // > 1 for Stanford and Hybrid models
6: // nrh is the nature of role-hierarchy. It is either Stanford or Hybrid
7: // rc is the number of elements a role is assigned to. It is either
8: // random or uniform
9: // uc is is the number of roles a user is assigned to. It is either
10: // random or uniform
11: // pc is is the number of permissions a role gets assigned to. It is
12: // either random or uniform
13:
14: We arrange roles in layers where the number of layers is dr and the
15: number of roles in each layer is nr/dr
16:
17: if rc is random then
18:     if nrh is Stanford then
19:         Each role at a specific layer is assigned to roles at a layer
20:         directly below it randomly
21:     else if nrh is Hybrid then
22:         Each role at a specific layer is assigned to roles at any layer
23:         below it randomly
24:     end if
25: else if rc is uniform then
26:     if nrh is Stanford then
27:         Each role at a specific layer is assigned to roles at a layer
28:         directly below it uniformly
29:     else if nrh is Hybrid then
30:         Each role at a specific layer is assigned to roles at any layer
31:         below it uniformly
32:     end if
33: end if
34: if uc is random then
35:     if nrh is Stanford then
36:         Each user is assigned random roles which are at the higest layer.
```

```
    else if nrh is Hybrid then
        Each user is assigned random roles which are at any layer.
    end if
else if uc is uniform then
    if nrh is Stanford then
        Each user is assigned uniform roles which are at the higest layer.
    else if nrh is Hybrid then
        Each user is assigned uniform roles which are at any layer.
    end if
end if
if pc is random then
    if nrh is Stanford then
        Each role at the lowest layer is assigned random permissions
    else if nrh is Hybrid then
        Each role at any layer is assigned random permissions
    end if
else if pc is uniform then
    if nrh is Stanford then
        Each role at the lowest layer is assigned uniform permissions
    else if nrh is Hybrid then
        Each role at any layer is assigned uniform permissions
    end if
end if
```

Categories	Parameters
Sessions	i. Total number of session initiations varies ii. Number of active sessions varies
Role Activation	i. Number of roles to be activated varies ii. Activate roles sharing the same permission at varying rates. iii. Activate roles at different levels of the role hierarchy. iv. Mutually exclusive roles are not activated at the same time.
Access Requests	i. Number of access requests varies ii. Perform access requests on random permissions available in the RBAC configuration. iii. Positive permissions are accessed more often.

Table 3.2: Session Profile Categorization

sessions, we mean sessions that have not been deleted yet after initiation. If the number of active sessions is reached, a fixed number of access requests is performed in the context of an active session. If the number of access requests is reached, we initiate new sessions. We repeat the process until the total number of session initiations is reached.

Role Activation For each session we initiate, the set of roles to we activate can either share the same permission, be located at different levels of the role hierarchy, or they could be roles that do not share the same parent role. In the latter case, roles are mutually exclusive in accordance to the concept of Separation of Duty [11].

Access Requests There is a fixed number of access requests performed in the context of active sessions. Access requests can either be performed on random permissions; or they can be performed on permissions for which the user is allowed to only since we anticipate that in most of the cases, users tend to request permissions for which they are allowed to more often.

Algorithm 7 is an algorithm we have implemented to generate session profiles for specific RBAC policies for our benchmark. We note the following method:

SessionProfile(ns, nsa, nrs, nr, nua, naa) – this method takes as input the number of sessions (ns), sessions per access request (nsa), roles per session (nrs), access requests (nua), and the nature of roles (nr) and access requests (naa). We record session initiations that may happen in lines 23-38 by activating nrs number of roles, which are determined according to nr. In lines 10-19, we record access requests to permissions, which are determined according to naa. Access requests happen when nsa is reached. We record the deletion of sessions if the number of active sessions is larger than nsa in lines 20-22, or if active sessions remain after ns is reached in line 41.

Algorithm 7 An algorithm for generating different session profiles for specific RBAC policies

Operation SessionProfile(ns, nsa, nrs, nr, nua, naa)

```
1: // ns is the number of sessions we initiate
2: // nsa is the number of sessions per access request
3: // nrs is the number of roles per session that we activate
4: // nr is the nature of roles.
5: // nua is the number of access requests we perform
6: // naa is the nature of access requests.
7:
8: count = 0
9: for i = 1 to ns do
10:     if count = nsa then
11:         // perform access requests
12:         for j = 1 to nua do
13:             if naa = 0 then
14:                 perform access requests to all permissions
15:             else if naa = 1 then
16:                 perform access requests to permissions a user is allowed to exercise
17:             end if
18:         end for
19:         record the access request with the permission and session
20:     else if count > nsa then
21:         delete one of the sessions created previously, and record it
22:         count−−
23:     else
24:         // perform a session initiation and activate a set of roles according
25:         // to nr
26:         for j = i to nrs do
27:             if nr = 0 then
28:                 activate roles that share the same permissions only
29:             else if nr = 1 then
30:                 activate roles at different levels of the role-hierarchy
31:             else if nr = 2 then
32:                 activate some junior roles but not all, making sure to not activate
33:                 parent roles which would activate other junior roles in order to
34:                 provide Separation of Duty
35:             end if
36:         end for
```

```
37:         record the session initiation with the user and set of roles
38:         count++
39:     end if
40: end for
41: record the deletion of the remaining active sessions.
```

Chapter 4

Evaluation of the three Data Structures

We implement a benchmark for RBAC policies and session profiles based on the categorization given in Chapter 3. We perform a statistically rigorous evaluation for the three data structures used by the SDP in Figure 1.3, and we compare the results.

Section 4.1 presents Georges et al's [13] approach that we use to evaluate the three data structures, and section 4.2 summarizes the evaluation results.

4.1 Evaluation Methodology

Meaningful empirical assessment is a significant challenge in computing [25]. For Java programs, non-determinism in making empirical observations can result from various factors, such as dynamic compilation and garbage collection. The methodology we adopt

overcomes such non-determinism and is statistically rigorous. It is based on the work of Georges et al. [13].

Java programs run within an instance of a Virtual Machine (VM). We collect the average time across multiple VM invocations, as there can be variation across such invocations. Within a VM invocation, we need to avoid skew from the effects of starting up the VM and reach what is called steady-state [13]. For each VM invocation, we determine the number of benchmark iterations that we need to perform by finding at least k consecutive steady-state values for which the coefficient of variation (CoV) is less than some preset value (we have chosen 2%). The value of k starts at some value (4, in our case) and increases so long as the CoV decreases, up to the threshold. We record the mean of the k values for each VM invocation. Our final benchmark time is the mean across all VM invocations.

To minimize the effects from garbage collection, we keep the heap size constant across VM invocations. Apart from the mean, we also compute confidence intervals. Our objective is for the confidence intervals to not overlap, which guarantees that with a certain confidence (95%, in our case), we can assert that the two values are statistically distinct. All the values we report and graph are statistically distinct from other values.

Algorithm 8 is an algorithm we have implemented to collect the evaluation results using Georges et al's [13] approach. We note the following methods:

Measurements(rp, sp, I, ds, k) – the method takes as input the RBAC policy (rp), the session profile generated for that policy (sp), the number of benchmark iterations (I), the data structure we intend to evaluate (ds), and the number of initial steady state values we intend to record (k). In lines 14-17, we calculate the mean of k timing measurements. In lines 19-23, we calculate the standard deviation of the k measure-

ments. In lines 25-32, we calculate the CoV to determine if the k measurements represent steady state values for I iterations. If they do, we record the mean of the k measurements. We increment k and collect new steady state values if the CoV can be minimized with a larger k. We record a new mean if k can be incremented to lead to a lower CoV.

ConfIntervals(means) – this method takes as input the means that we record for each VM invocation of the method **Measurements**. In lines 5-9, we evaluate the mean of all VM invocation measurements. In Lines 11-15, we calculate the standard deviation of the measurements, and we calculate the confidence invervals in lines 17-19.

We have conducted our experiments on an isolated Intel dual core E8400 PC that runs at 3 GHz, has 3.5 Gbytes of RAM and runs the Ubuntu Linux operating system. Our Java version is 1.6.0_18, and we run the OpenJDK Runtime Environment.

4.2 Comparison of Final Results

The interface we use to represent the architecture in Figure 1.3 is implemented in Java by Komlenovic [19]. Section 4.2.1 and section 4.2.2 present an analysis of the time- and space-efficiency of the three data structures.

4.2.1 Time Efficiency

We present our results for time efficiency in Table 4.1. In each dataset we have 2500 users, each authorized to different numbers of roles and permissions. We have 100 roles in total, and 100 permissions.

Algorithm 8 An algorithm for finding k consecutive steady state values when evaluating a data structure for the lowest possible CoV starting from 0.02 and I iterations for each VM invocation, then evaluating the mean and confidence intervals of m VM invocations

Operation Measurements(rp, sp, I, ds, k)

1: // rp is the RBAC policy used in our evaluation
2: // sp is the session profile generated for rp
3: // I is the total number of benchmark iterations
4: // ds is the data structure, which is either Authorization Recycling,
5: // the Bloom Filter, or the Cascade Bloom Filter
6: // k is the number of steady state values
7:
8: we calculate and record $time_1 \dots time_I$ where $time_i$ is the total access request time when using ds for rp and sp
9:
10: $CoVtmp = 0.02$
11: **for** i = I **to** 1 **do**
12: $mean = 0$
13: // calculate the mean
14: **for** j = 0 **to** k **do**
15: $mean = mean + time_{i-j}$
16: **end for**
17: $mean = \frac{mean}{k}$
18: // calculate the standard deviation
19: $stdev = 0$
20: **for** j = 0 **to** k **do**
21: $stdev = stdev + (time_{i-j}$ - $mean)^2$
22: **end for**
23: $stdev = \sqrt{\frac{stdev}{k-1}}$
24: // calculate the minimum possible coefficient of variation
25: CoV = $\frac{stdev}{mean}$
26: **if** CoV < CoVtmp **then**
27: record $mean$, or overwrite the previous recorded $mean$
28: CoVtmp = Cov
29: i++
30: k++
31: **end if**
32: **end for**
33: **if** no steady state values have been found **then**
34: **return** $false$

35: **else**
36: **return** *true*
37: **end if**

Operation ConfIntervals(means)

1: // means is the set of means $mean_1 \dots mean_m$ where $mean_i$ the
2: // output of the ith VM invocation to Measurements
3:
4: // calculate the mean
5: $mean = 0$
6: **for** i = 1 **to** m **do**
7: $mean = mean + mean_i$
8: **end for**
9: $mean = \frac{mean}{m}$
10: // calculate the standard deviation
11: $stdev = 0$
12: **for** i = 1 **to** m **do**
13: $stdev = stdev + (mean_i - mean)^2$
14: **end for**
15: $stdev = \sqrt{\frac{stdev}{m-1}}$
16: // calculate for 95% confidence intervals
17: interval = $1.96 \times \frac{stdev}{\sqrt{m}}$
18: lower_interval = mean - interval
19: upper_interval = mean + interval

		Auth. recycl.	Bloom filter	Cas. Bl. filter
Bursty	Stanford	2928.80	56.03	18.07
	Hybrid	94.07	60.15	32.40
	Core	10.18	50.41	29.94
Uniform	Stanford	1220.43	53.73	22.00
	Hybrid	49.73	55.57	25.54
	Core	4.44	55.62	26.16

Table 4.1: Average access request times in μs. The averages are across number of total session initiations from 2 through 15, for a given RBAC policy that comprises 2500 users, 100 roles and 100 permissions. For the Stanford RBAC policy, we have adopted 5 layers, which is the maximum in Table 3.1. For the Hybrid RBAC policy, the depth varies between 1 to 5. We give the standard deviations for the bursty case in Figure 4.1.

Our objective is to understand the behavior of each data structure as we vary the number of sessions and sessions per access request. Consequently, we consider from 2 through 15 sessions, and both bursty and uniform arrivals for the sessions. By bursty arrival, we mean that session initiations are interspersed with relatively long "quiet" periods in which we have no session initiation. In the interim, we have access requests for the sessions that exist. In uniform session arrival, session initiations are uniformly interspersed with access requests. We conjecture that bursty arrivals are likely with sessions that are directly used by humans, and uniform arrivals are possible if there are automated processes with which sessions are associated. The results are displayed in Figure 4.1.

We vary three parameters in our experiments: the number of roles per session, the number of permissions per session and the nature of RH (Stanford, Hybrid and Core). Figures 4.1 shows the impact of the last attribute on time efficiency, and 4.2 shows average access request times in μs for Core RBAC, for which the number of roles and permissions range from small (10) to large (10,000). Such numbers are consistent with Table 3.1.

We give an analytical summary of the results we have generated in the following sections.

Number of Sessions per Access Request (Bursty vs. Uniform)

We observe from Table 4.1 that none of the data structures, except Authorization Recycling, is impacted by the session arrival rate (burst vs. uniform). The reason is that in Authorization Recycling, all the work is during access request; session initiation does not involve any exchange from the PDP to the SDP (except validation of the initiation). Consequently, Authorization Recycling can be impacted by bursty session arrival, which results in a number of access requests in short periods.

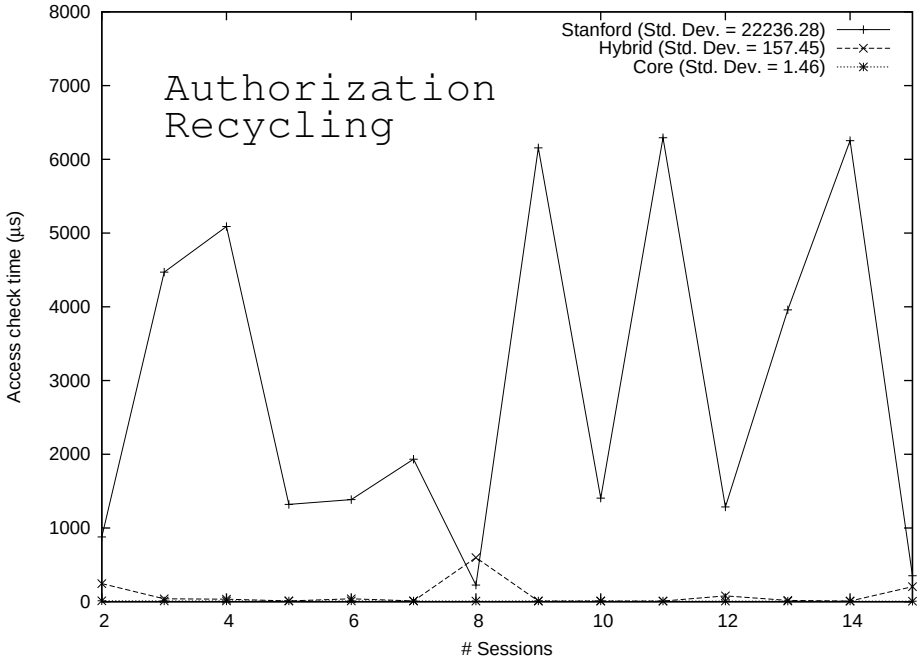
Total Number of Sessions

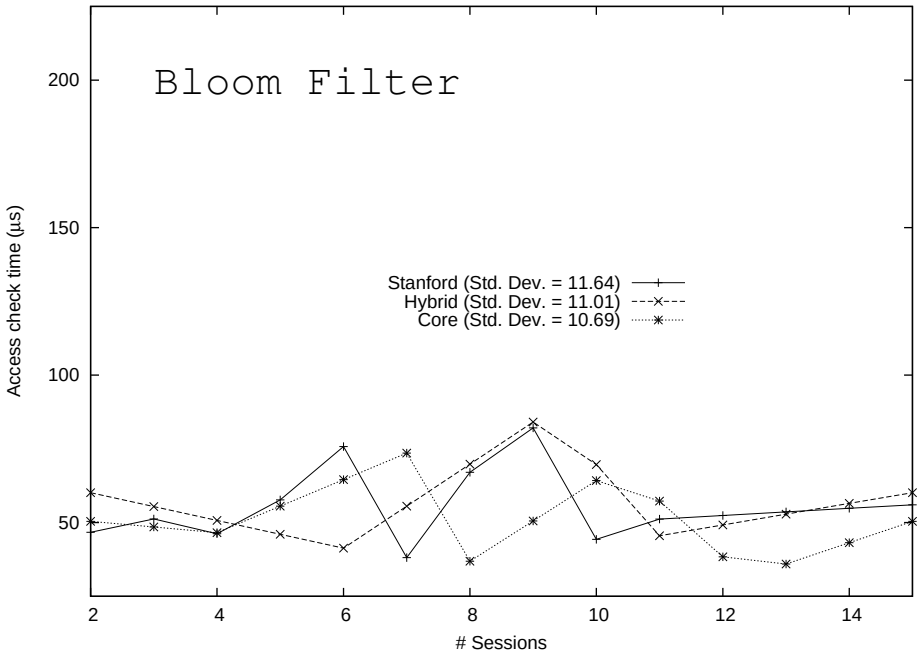
The graphs in Figure 4.1 show the impact of the number of sessions on each data structure. We observe that all three data structures are resilient to an increase in the number of sessions from the standpoint of time-efficiency. That is, the access request time does not necessarily grow with the number of sessions. We expect this to be the case, so long as the PEP/SDP is not stressed by adding too many sessions. None of the data structures has an access request algorithm whose time-complexity is parameterized by the number of sessions.

It is not our objective to stress a PEP/SDP by considering large numbers of sessions. Indeed, the number of sessions a PEP/SDP can support without significant impact on its performance depends on its resources such as hardware. Our objective is gain broader insights into the three data structures, notwithstanding the resources available to a PEP/SDP, assuming some realistic model of computation (the "Random-Access Machine" model, for example [7]).

Efficiency

The Cascade Bloom Filter and the Bloom Filter are time-efficient data structures. The major overhead with them is the computation of the hash function (in our implementation, this is the cryptographic hash function, SHA-1 [29]), and searching a set in the worst case. Authorization Recycling is efficient for Core-RBAC, but its performance degrades when we add a hierarchy. The reason is that the first time a permission is accessed, the SDP must communicate with the PDP; this must happen for every permission that is accessed.


Authorization
Recycling
Stanford (Std. Dev. = 22236.28)
Hybrid (Std. Dev. = 157.45)
Core (Std. Dev. = 1.46)
Access check time (µs)
Sessions
8000
7000
6000
5000
4000
3000
2000
1000
0
2
4
6
8
10
12
14


Bloom Filter
Stanford (Std. Dev. = 11.64)
Hybrid (Std. Dev. = 11.01)
Core (Std. Dev. = 10.69)
Access check time (µs)
Sessions
200
150
100
50
2
4
6
8
10
12
14

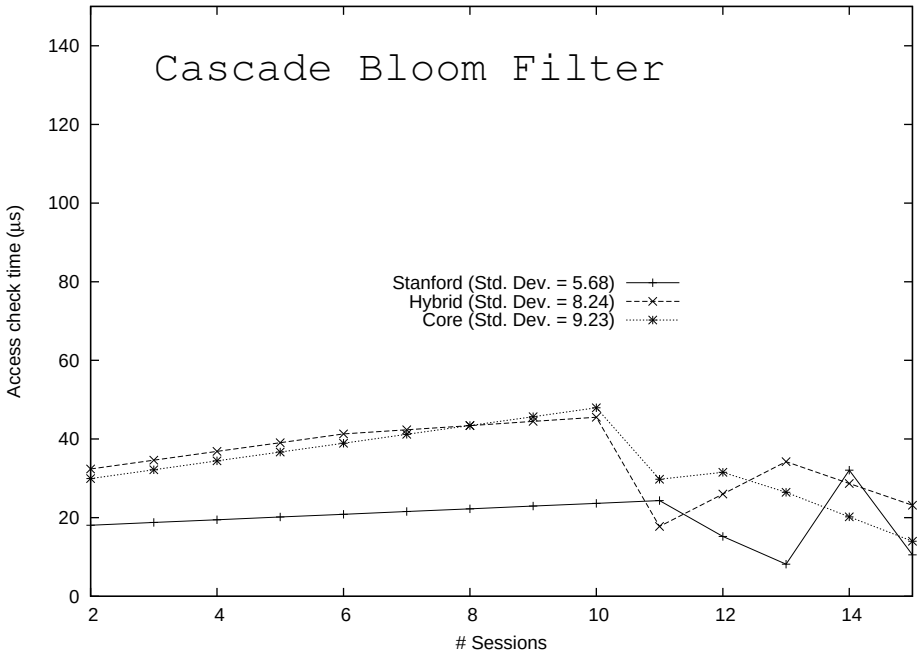


Figure 4.1: Average access request times in μs and the corresponding standard deviations for the three data structures as the number of sessions changes, for the three different kinds of role hierarchies.

Jitter

By jitter, we mean the variation in access request times as the number of sessions changes. We can quantify this as the percentage error in the mean; that is, the ratio of the standard deviation to the mean. We observe from Figure 4.1 that this is quite high for the Cascade and Bloom Filter, and very high for the Stanford RH for Authorization Recycling. The Cascade and Bloom filter are affected by the heterogeneity of the permissions; if the union of permissions to which all sessions are authorized is larger, this can result in a deeper Cascade or a larger set of false positives that must be maintained explicitly. Authorization Recycling is affected by the heterogeneity of the roles that a user may activate in a session. In our datasets, a user is directly assigned to the same number of roles across each of the Stanford, Hybrid and Core policies. Consequently, there is more heterogeneity in the roles that a user may activate in the Stanford policy than in the other two.

Role Hierarchy (Stanford vs. Hybrid vs. Core)

Table 4.1 and the graphs in Figure 4.1 show the impact of Stanford vs. Hybrid vs. Core as the choice for RH. Only for Authorization Recycling do we see an impact from the choice of RH. A deeper RH gives a user more choices of roles he may activate during session initiation. This is reflective of our dataset – a user is directly assigned to the same number of roles for all three of the Stanford, Hybrid and Core RBAC policies. However, in the Stanford policy, he is authorized to more roles as a result of the deep RH. Consequently, the size of Cache^- and Cache^+ is larger.

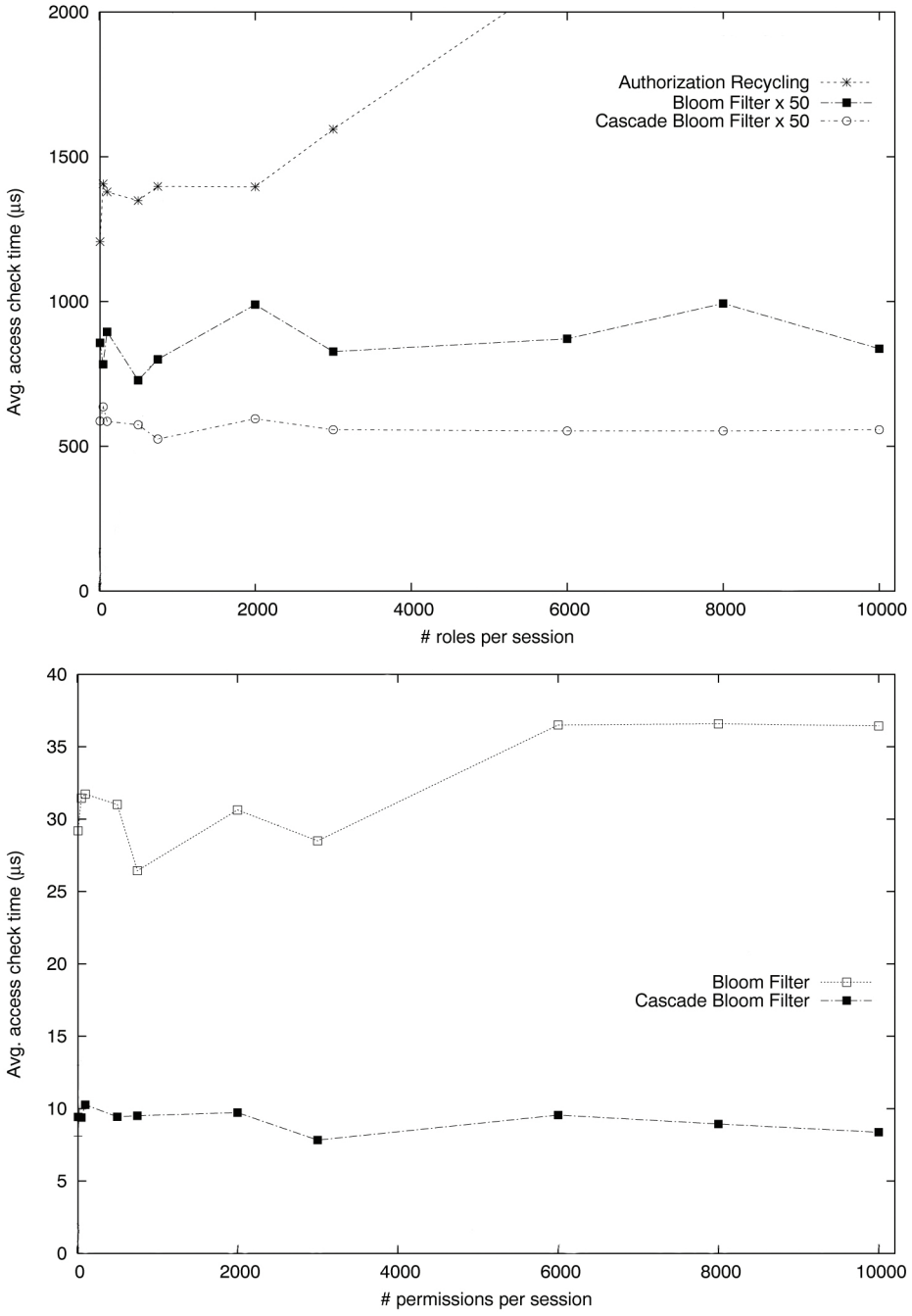


Figure 4.2: Time-efficiency for small (10) to large (10,000) numbers of roles and permissions in a session. In the lower graph, we do not plot Authorization Recycling as the numbers are much larger than the ones for the other data structures.

Scalability

We observe from Figure 4.2 that Authorization Recycling fares somewhat poorly as we need to store almost every role for each permission, and this results in it being linear in the number of roles. For the Cascade and the Bloom Filter, the time for access request is independent of the number of roles in a session. In this respect, they scale well with the number of roles.

The Cascade and the Bloom Filter scale well also with the number of permissions, as Figure 4.2 indicates. Also, for the the optimal values of the number of indices and the number of hash functions changes with the number of permissions. (It may decrease for the Cascade Bloom filter owing to the negation bit.) Notwithstanding this, up to 10,000 permissions, these issues appear to have no tangible impact on the time efficiency of these data structures. We do not plot Authorization Recycling in the graph for permissions in Figure 4.2 as the numbers for it are much higher than for the other data structures. It scales poorly with the number of permissions, as the number of entries in the two sets is linear in the number of permissions in a session.

4.2.2 Space Efficiency

In this section, we analyze the space-efficiency of the three data structures. We base our assessment on what we have observed from our implementations, and an analysis of the algorithms we implement.

Authorization Recycling is linear in the number of sessions. However, it can be quadratic in the number of roles and permissions, in the worst case. The reason is that an entry in the Cache^- or Cache^+ is a role set-permission pair.

The Bloom Filter and the Cascade Bloom Filter are non-constant in space relative to the number of sessions and the number of permissions per session. The reason is that the optimal values for the number of indices and the number of hash functions changes with the number of sessions and permissions. For the Cascade Bloom Filter, the number of indices may decrease with an increase in the number of sessions or permissions per session as a consequence of the negation bit (see Section 2.1 and Figure 2.8). Consequently, the relationship between space and the number of sessions or permissions per session is a step function. The Cascade and the Bloom Filter are agnostic to the number of roles per session.

In Figure 4.3, we present graphs that capture the above discussion. The graphs have been generated based on our implementations.

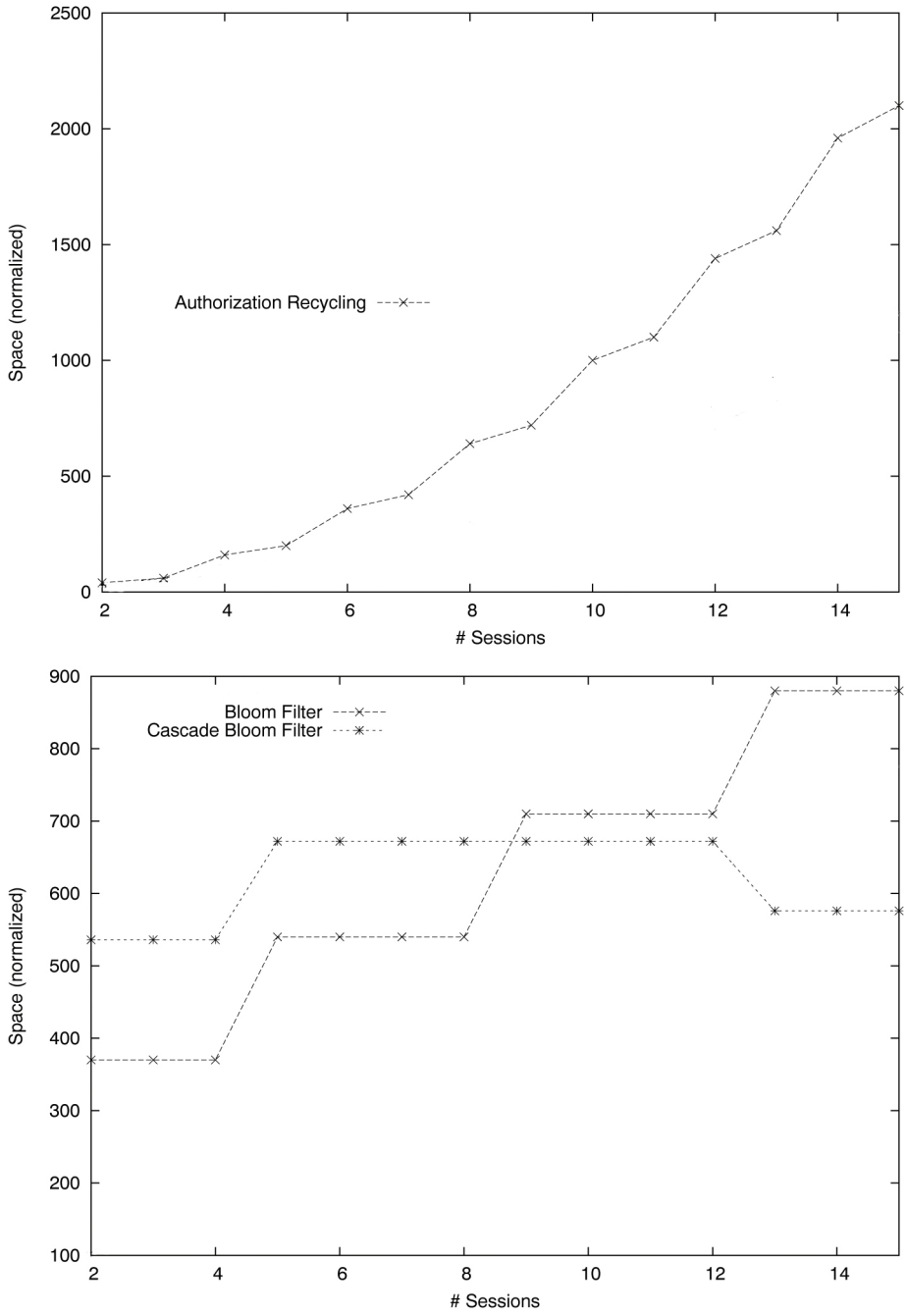


Figure 4.3: Space inefficient (upper figure) vs. space efficient (lower figure) data structures using a constant growing number of roles and permissions per session

Chapter 5

Conclusion

We have provided a sound and reliable performance evaluation of the Bloom Filter, the Cascade Bloom Filter, and Authorization Recycling for distributed access enforcement in RBAC systems. The following sections are organized as follows: Section 5.1 summarizes our evaluation results, and section 5.2 presents possible improvements that could be done in the context of access enforcement.

5.1 Summary

The Cascade Bloom Filter is more efficient than the Bloom Filter and Authorization Recycling. We observe from Figures 4.1 and 4.2 that the Cascade is the most time-efficient implementation. It is resilient to changes in the number of sessions, and the number of roles and permissions per session. We observe from Figure 4.3 that the Cascade is the most space-efficient implementation as the number of sessions increases. Therefore, we conclude that the Cascade Bloom Filter must be used for distributed access enforcement in RBAC

systems.

5.2 Future Work

Real world enterprises may comprise a large number of users that scales to more than a million. We anticipate that these enterprises will implement an RBAC model to perform access enforcement. As hardware capabilities improve, it is essential to evaluate the three data structures for a larger number of users, roles and permissions, and for different hardware configurations. The existence of the Cascade Bloom Filter that encodes any possible set needs to be proved. We need to broaden our benchmark by considering different hardware models for our evaluation. Finally, we need to perform our evaluation by stressing the PEP/SDP and observing the output. The output varies depending on the hardware model.

Bibliography

[1] Global 500. Fortune Magazine 2010. Available From `http://money.cnn.com/magazines/fortune/global500/2010/`. 39

[2] R. Anderson, R. Bohme, R. Clayton, and T. Moore. *Security Economics and the Internal Market*. Feb. 2008. 1

[3] E. Bertino, A. Kamra, E. Terzi, and A. Vakali. Intrusion Detection in RBAC-administered Databases. *Proceedings of the Applied Computer Security Applications Conference(ACSAC)*, 21, December 2005.

[4] B.H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970. 11

[5] C. Blundo and S Cimato. A simple role mining algorithm. *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 1958–1962, 2010. 38

[6] K. Borders, X. Zhao, and A. Prakash. CPOL: High-performance policy evaluation. *in Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pages 147–157, 2005. 5, 35

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, September 2009. 53

[8] Personal Communication. Open Text Corporation. Aug. 2010. 4

[9] J. Crampton. On permissions, inheritance and role hierarchies. *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS-10)*, pages 27–31, October 2003. 39

[10] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000. 12

[11] D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, 2003. 1, 8, 38, 39, 43

[12] M. Frank, A.P. Streich, D. Basin, and J.M. Buhmann. A Probabilistic Approach to Hybrid Role Mining. *Proc. 16th ACM conference on Computer and Communications Security (CCS)*, pages 101–111, 2009. 38

[13] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. *Proceedings of OOPSLA'07*, pages 57–76, May 2007. x, 5, 47, 48

[14] V. Hu, D. Ferraiolo, and D. Kuhn. Assessment of access control systems. *Technical Report NISTIR 7316*, September 2006.

[15] M. Jafari, A.H. Chinaei, K. Barker, and M. Fathian. Role Mining in Access History Logs. *Journal of Information Assurance and Security*, 2009. 38

[16] J.B.D. Joshi, E. Bertino, and A. Ghafoor. Temporal hierarchies and inheritance semantics for gtrbac. *SACMAT '02: Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 74–83, 2002.

[17] A. Kern. Advanced features for enterprise-wide Role-Based Access Control. *Proceedings of the 18th Annual Computer Security Applications Conference*, pages 333–343, December 2002. 38

[18] A. Kern, M. Kuhlmann, A. Schaad, and J. Moffett. Observations on the role life-cycle in the context of enterprise security management. *7th ACM Symposium on Access Control Models and Technologies*, June 2002. 38

[19] M. Komlenovic. A Platform for Assessing the Efficiency of Distributed Access Enforcement in Role-Based Access Control (RBAC) and Its Validation. *MASc Thesis*, December 2010. 49

[20] M. Komlenovic, M. Tripunitara, and T. Zitouni. An Empirical Assessment of Approaches to Distributed Enforcement in Role-Based Access Control (RBAC). *Accepted to appear, ACM Conference on Data and Application Security and Privacy (CODASPY)*, February 2011. iv, 7

[21] Y.A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core Role-Based Access Control: efficient implementations by transformations. *PEPM'06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pages 112–120, May 2006. 8, 35, 36, 38, 40

[22] R. McRae. The Stanford Model for Access Control Administration. Stanford University, 2000 (unpublished).

[23] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo. Mining roles with semantic meanings. *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2008. 38

[24] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo. Evaluating Role Mining Algorithms. *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, pages 95–104, 2009. 6, 8, 38, 40

[25] T. Mytkowicz, A. Diwan, M. Hauswirth, and P.F. Sweeney. Producing wrong data without doing anything obviously wrong! *Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, pages 265–276, 2009. 5, 47

[26] R.S. Sandhu. Future Directions in Role-Based Access Control Models. *Proceedings of the International Workshop on Information Assurance in Computer Networks: Methods, Models, and Architectures for Network Security*, 2001.

[27] A. Schaad, J. Moffett, and J. Jacob. The role-based access control system of a european bank: A case study and discussion. *proceedings of ACM Symposisum on Access Control Models and Technologies*, pages 3–9, May 2001. 38

[28] J. Schlegelmilch and U. Steffens. Role mining with ORCA. *Symposium on Access Control Models and Technologies (SACMAT).*, June 2005.

[29] F. I. P. Standards. Secure hash standard. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`, 2002. 53

[30] M.V. Tripunitara and B. Carbunar. Efficient access enforcement in distributed Role-Based Access Control (RBAC) deployments. *SACMAT 09: Proceedings of the 14th ACM symposium on Access control models and technologies*, pages 155–164, 2009. 5, 14, 17, 18, 26, 27, 31, 35, 36, 38, 40

[31] J. Vaidya, V. Atluri, and J. Warner. Roleminer: mining roles using subset enumeration. *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, pages 144–153, 2006. 38

[32] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in RBAC systems. *Proceedings of the thirteenth ACM Symposium on Access Control Models and Technologies (SACMAT).*, pages 63–72, June 2008. 4, 5, 27, 30, 31, 32, 35, 36, 38

[33] Q. Yao, A. An, E. Terzi, and X. Huang. Finding and Analyzing Database User Sessions. *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005. 40

[34] D. Zhang, K. Ramamohanarao, T. Ebringer, and T. Yann. Permission set mining: Discovering practical and useful roles. *ACSAC'08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 247–256, 2008. 38

[35] D. Zhang, K. Ramamohanarao, S. Versteeg, and R. Zhang. Rolevat: Visual assessment of practical need for Role-Based Access Control. *ACSAC*, pages 13–22, 2009. 38